UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Mobile app of a shopping list

Final Degree Project

Author: Pau Trepat Segura
Director: Enric Mayol Sarroca
Specialization: Software Engineering

June 3, 2016

Resum

Avui dia hi ha milions d'aplicacions que donen resposta a la majoria de les nostres necessitats i ens faciliten el dia a dia. No obstant, no totes les nostres necessitats troben resposta en les aplicacions que hi ha al mercat. Aquest treball final de grau és fruit d'un projecte personal que vol resoldre la necessitat de tenir una llista de la compra compartida entre diverses persones. En diferents casos (amics que comparteixen pis, parelles o famílies) existeix la necessitat de tenir una llista comuna dels productes que s'han de comprar. Això no és possible amb les aplicacions que existeixen i la solució d'apuntar-ho en un paper, com a informàtics, no ens satisfà.

La segona qüestió que ens impulsa a dur a terme aquest treball és la falta de coneixements sobre desenvolupament de *software* més enllà d'entorns controlats en pràctiques, on les males decisions només es paguen en forma de mala nota. Som conscients que els mals dissenys de *software* i les males pràctiques s'acaben pagant i el seu preu pot arribar a ser molt alt. Volem que aquest treball sigui un exercici d'interiorització de com desenvolupar codi correctament pensant amb compte cada decisió i seguint les *best practices* habituals de cada context.

Abstract

Nowadays there are a lot of apps which give us a response for every need we could have, and make life easier. Nevertheless, not every need is covered by the apps that are currently in the market so this final project wants to solve one of these uncovered needs. Specifically, the project tries to solve the necessity to have a sharing buy list between some people (colleagues, relatives, couples, etc.) who actually have the necessity to have a common list of the products that they want/have to buy. Unfortunately, there is no app available with this functionality, at the moment, and the simple solution of writing the list in a paper is so basic for an informatics engineer.

Moreover, another motivation for this project is the lack of knowledge about development of software in not prepared environments. As students we are accustomed with exercises where the environment is really controlled, because they are prepared to teach us something. In this case, an error is punished only with a bad grade. On the other hand, outside university, the bad designs of software and the bad practices in codding might suppose a very high cost, in terms of time and money. For this reason, we want this project could be and exercise of interiorizing how to develop code in the right way, thinking carefully every step and decision and following software best practices in each situation.

Acknowledgments

Thanks to my family for giving me the chance to study what I want and love.

Especially to my father Jordi, because although he does not know a lot about informatics, have made all the linguistics corrections of my essays and projects during the degree. Thanks for your time and dedication.

Thanks to my mother Maria Àngels to have listened heroically some presentations and speeches practices that neither I would have endured.

Thanks to my brother David to allow me to bother him with the light and the sound of the computer at late hours at night. I hope that from now you will be able to sleep quietly.

Thanks to my workmates who have allowed me to grow professionally and personally. Especially to Joan López with who I have had long conversations about design patterns, software architecture and best practices. For all the *ñapas* we let behind us.

Thanks to Enric Mayol, who always has had a moment for me when I have asked for a meeting, which have been a lot of times.

Thanks to all the teachers whom I could ask questions about this project. Thanks for a time you do not have to give to me. Thanks also to all the teachers I have had during the degree and have had dedicate to me some of their time, patience and empathy. Thanks for transmitting to me your passion in what you do.

Last but not least, thanks to Aixa. For the time, patience, support, comprehension, encouragement and, especially, interest which has helped me to go ahead with this project and many others.

Index of contents

List of Tables

List of Figures

1 Introduction

1.1 Context

To understand this project and its goals it is necessary to offer a scope of the situation where we were at the moment we considered do the project.

The first question we have to ask is "why an app related with a shopping list?" finds the answer to any problem related with shopping lists, but as we will see in the next pages, any of the actual apps is able to solve the needs that exist nowadays. But, which are these needs we have detected?

Usually people who share home meet each other only a few times during the week, and there are basic products which are depleted before the end of the week. In front of this situation they could buy these products on Saturday or they could buy them in day to day. If they opt for the second option, we could have three problematic situations: firstly, that two or more members of the house buy the same product. As the fridge has a limited capacity this could bring a problem of lack of space. Secondly, that none of the members of the house buy the product thinking that some other member would do it. Despite they could buy the product later, maybe they are not able to do it because any store is open, and this brings us to do not have the product we need. Thirdly, the ideal situation which consists in that only one of the members buys the product taking advantage of the fridge capacity and disposing of the product.

This is the problem of every day, where the quantity of products to buy is low but its priority is high. The second and most common problem is to be able to list and save the products that at the end of the week we'll need to buy but they are not urgent and it can be stored without problems of space, such as milk or water, because its space of conservation is not limited to refrigerators. In addition to want to store them, is also very useful to be able to edit them simultaneously because it allows that different people can buy simultaneously the same list; put us in context. Throughout the week we have been adding what we needed to buy and when the shopping day arrives there are a big number of products and we want to go as fast as possible. If the application allows a simultaneous edition we can shopping the different products from the list at the same time without buying repeated products, saving time and efforts.

Aquesta és la problemàtica del dia a dia, on la quantitat de productes a comprar és baixa però la seva prioritat és alta. La segona problemàtica i més habitual és la de poder llistar o guardar els productes que al final de la setmana necessitarem comprar però que no són urgents o es poden emmagatzemar sense problemes d'espai, com la llet o l'aigua, ja que el seu espai de conserva no es limita al frigorífic. A més a més de voler-los emmagatzemar també és de gran utilitat poder-los editar simultàniament ja que això permet que diverses persones puguin comprar la mateixa llista alhora; posem-nos en context. Al llarg de la setmana hem anat afegit què necessitàvem comprar i arribat el dia de la compra hi ha un bon nombre de productes i ens interessa anar el més ràpid possible. Si la aplicació permet una edició simultània podrem comprar els diferents productes de la llista per separat sense comprar productes repetits, estalviant temps i esforços.

Because basically this problem is shared by people around us and we want to implement an agile methodology, to develop functionalities under demand we'll consider these people interested in the app, our customers.

1.2 Problem to solve

Based on the first context described above, there is a situation where a set of individuals (known as members) belong to a group (either a couple or a group of students that shared a flat) want to share information related with products they have to buy, which we call shopping list. Also in certain cases they want to share immediately the need to buy some products. This is the first situation we want to solve.

The other problem which derives from the second context is the lack of knowledge, good practices and methodologies which are necessary out of the academic environment when we have to develop software. But it is not only a theoretical issue (the lack of knowledge) but is more important: is a practical problem. Everybody is capable of sitting in front of an exercise which has been proposed in a theoretical frame and solve it. However, the difficulty lies in the analysis of the problem, the approach of different solutions and finally to take a decision, being aware that in the long run it could become in a heavy technical debt.

In this project we will emphasises this second problem because we consider that the learning of this best practices, design of accurate solutions and methodologies has a high value. Do not matter the technology used or the scope, what is important is to have interiorized all this concepts and practices which will allow us to contribute with a differential value.

1.3 Personal motivations

At the end of my degree and working part time, we have beginning to face with reality. There is no similarity between the projects we have to deliver during the degree, neither the goals nor the way to evaluate the results. Now, there is not good or bad choices, there is better or worse decisions, but especially, they start to think about the technical debt[1], a concept which refers to all the prices that we pay for our bad choices thought a project, and usually increases with time. This is an unknown concept during the degree because never a project has a length higher than 4 months. However, the software used by the companies may have a long "life" and is in the course of this time that the technical debt arises.

After almost a year of having worked in two projects, we have realized which are the costs and benefits of making things in a proper way, taking the time that it needs; the price of making things quickly to have a functionality ended; and finally which are the problems that has to face the person which has to maintain a system which has not been correctly designed. So, within six month to enter to the labor market as graduates, we would like this last project was a small collection of best practices and thoughtful choices, and not end delivering a project full of functionalities and extensive in code. We would like to deliver a project where each decision, every applied pattern and every contribution and advanced done is something to be proud and which can be a reference for future projects.

Giving these experiences in the software development, in this project we will bet on develop quality software, taking care of the architecture design and the implementation and trying to document ourselves about best practices to follow in order to achieve that the technical debt of this project will be as low as possible.

2 State of the art

Once we have seen which is the problematic related with the sharing of shopping lists, we have done a study about what there are nowadays on the app market that can solve this problem, from ideas already implemented on the market to what we can offer us as a differentiated value.

About the problematic related with best practices and methodologies, it is impossible to understand all the literature that it is available, so we have opt to describe the ones we have used during the project in a chapter, lately.

Focusing on the android market, we have found that there are several applications already functioning. Nowadays the most popular is *Out of Milk Shopping List* with 5 million of downloads, very far of other applications that have a mean of only 1 million of downloads. In front of this situation we have download several ones to try it but we will only include two in this state of art: *Out of Milk Shopping List*, because of its popularity, and *myShopi*, because it is one which have 1 million of downloads and gave us some new ideas.

2.1 *myShopi*

With a very different format, both in the visual aspect and the navigation, this app assumes an organization of stores where you can buy the product, which have gave us a new idea: add in our app the field of where to buy the product. It seems a simple thing with no complications, but it is not. Talking with potentials users of the app they have agreed they are some products that they have not bought in any other place so organize the shopping lists by supermarkets believe would help them to do shopping efficiently if they have to do it in different supermarkets.

However we have detected some functionality which makes harder using the app. For example, there is the possibility to have two shopping list of the same supermarket (Figure 1) or the possibility of mix in a list two different supermarkets. We have considered that it is not a good option to must to distribute the products by supermarket, because does not allow us to have a common list.

Besides this observations related with the establishments, we saw that it makes an aggregation by type of products when it showed them. It gave us an idea about how to classify and distribute by category the products: vegetables, groceries, drugstore, etc. This seems to us an excellent idea, but as the other one, complex to implement.

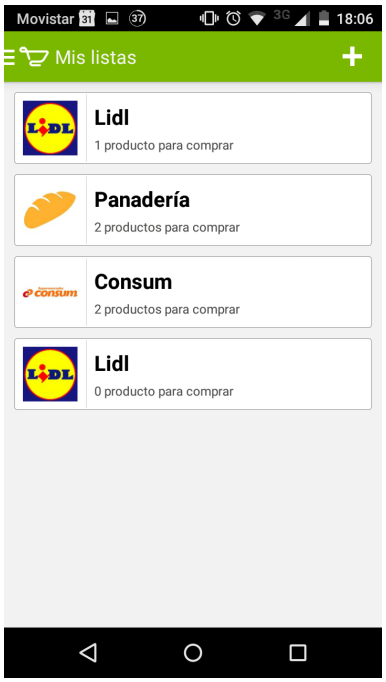

Figure 1: Screen of the store selection myShopi.

The last thing we observed, that was repeated in other apps was the possibility to introduce a products through the bar code, an agile way to interact with the app, which makes easier the users life.

2.2 *Out of Milk Shopping List*

It is the most popular app, in user terms. Its main screen is simple as we can see in the Figure 2.

This interface offers to the user useful information like as the accumulated total price or the products that he/she has already introduced in the shopping list. Moreover, besides of the keyword as a traditional way to introduce the products, it allows to add products using the voice (the microphone symbol) and with the lecture of a bar code.

The interface is simple and very usable, but the usability becomes complicated with the navigation. The app has two menus, one on the left and one on the right, one that allows the user to see the different shopping lists and get access to configuration (the left menu) and the other that allows you to access to certain information about the configuration of the shopping list which is already in the screen. What is the problem? The user can arrive to the same point using either the left or the right panel, which can confuse the him/her.

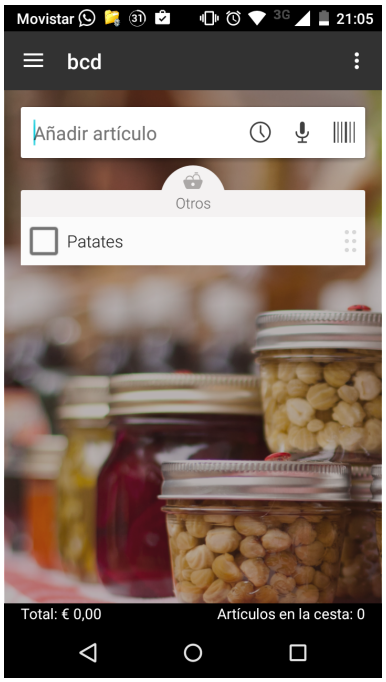

Figure 2: Home screen Out of milk.

Analyzing this app has brought up the first problem of our project. *"Out of Milk Shopping List"* offers the possibility to share the shopping list. However looking at the comments of the app at *Play Store*, there are several critical opinions because of the problems with the implementation of the functionality. We have decided to run some tests, and we have seen that the app does not work well, with some products that appear and disappear. For this reason, we have decided to continue with the project because some of the main needs –be able to share to shopping list- are not solve by the app.

This application has given us the idea of having several shopping lists and allows interacting between them, which could be very useful for the user.

3 Solution proposed

The solution of the problem with the shopping list is to create a system that allows having the information always available, so the information will have to be at the cloud. We have decided to create an API REST as interface to save and recover the information needed. Besides the functionalities of the list itself, the app will also have the user management and everything related with the user authentication on the API. About the user interface, a prototype of the app on android will be developed in order to being able to interact with the API. The processes or specific actions of the business logic will be defined as soon as our clients will ask for them. However, our initial premises are what we have already mentioned: at least the app must allow to have shared shopping lists between users. The other interactions, like being able to move products between lists or being able to connect to services of third parties will be left to the choice of the clients, because they are potential users of our app.

The solution to the second problem is more complex. The knowledge we want to obtain are the result of years of wanting to face several problems, and it is obvious that a final degree project cannot cover all of them. However our purpose is that on every step and on every obstacle we will have to face, will apply the best possible solution, always with the goal of making a software easy to change and simple to maintain, despite this would be in detriment of efficiency. We have decided to adopt this way of working because a person that has worked a lot of years in the information sector said this thoughts: "On the 99% of the cases, software that have efficiency and performance problems, you can solve it putting more computation resources, but a bad designed software, when you have to make some modifications, you will face a cost to fix it, maybe one hundred times higher than purchasing powerful machines".

Mainly, we are worried about two aspects:

To create a product independent of frameworks, data base engines or others. We want to follow the philosophy *Your application is not your framework*[2] which talks about to reach low coupling between our business or domain layer of everything related with the framework. We want an app which independently on what we support to develop it, change the context be relatively simple. We do not want our business attached to a serial of supports because the day that one of them died, our business will died too.

The second aspect is easy code maintenance. The idea is that if this project needs to be developed by another developer do not request excessive efforts for him to do it. In order to accomplish that, we have to having into consideration some best practices such as *Naming convention*[3] and a correct organization of the functions when we write the code. Other best practices in order to make easy the maintenance would be the application of the SOLID[4] principles, and mainly the implementation of tests. This would make easy the detection of errors before get the product to the market, and if we opt for TDD[5] when we develop the tests, would be easier the choice of better patterns of designs.

Independently of the technological and functional solution of our project, we will have to accomplish these two aspects. We do not think about a little work design but functional, because, as we have mentioned, our transversal goal is to do a deployment of best practices, accurate design and use of the properly design for each problem.

4 Development Plan

4.1 Goals

In the section about problematic to solve, we could see that there are two situations we want to solve. One which is very enclosed and simple to measure- the one about shopping lists- and another one more abstract and transversals, which is the intention to think carefully about each one of the solutions to develop. This second one can be seen reflected in the analysis of the problems, the choice of patterns and the design of the solution. We were aware this second goal could produce and overrun in terms of time in the development part, but as we have mentioned before, we preferred to spend an extra time now and not be forced to spend a lot of time in the future solving problems derived of bad choices in the past.

4.1.1 Best practices and correct software design

In order to make a better design of our system we support on:

- Use of SOLID [4] principles: one of the most frustrating situations that a developer can experiment is that he/she wants to apply the SOLID principle in a code inherited but he/she cannot do it in the first moment because of a bad design. A bad design does not allow us (without making refactor) to apply them. We will avoid this situation, if at the beginning of the project we apply them and we facilitate the way to futures developers.
- TDD [5]: if it is frustrating to have to solve problems when exist a test, it is much worst when it does not exist. The tests allow us to assure that our code works without have to make all the deployment. If also we write this test using TDD we obtain advantage (among others) to detect errors in the design before the implementation.
- Hexagonal architecture [6]: this goal answers to our will of follow the philosophy *Your application is not your framework*. In the future we will explain how to do it.
- Document the creation of a virtual machine (if it is possible automatization): one of the best practices none related with the development itself is the automatization of processes which manually could induce to errors. Our intention is to document the creation of a virtual machine because any person could explain easily the environment, because at least there will be an equal environment for development and production.
- Automatize the creation of the database of development and production: following with the previous philosophy our intention is that with a couple of scripts be able to create the database of production and development.

- Prepare the API and the application to support a possible new version: one of the most important problems of the development of an API is the retro-compatibility. If any new functionality implies a change on the API's answer and this was not compatible with old versions we would have two solutions: the users with this versions will have to update the application or find a solution where both the new and the old versions of the app continue working. We have not already decided which option will choose but we will prepare the server to be able to make both.

- Use, if it is possible, the REST standards: although the API will be for private consume and not open to third parties, we opt to follow these standards.

- Follow the best practices regarding the development on applications: the mobile applications have nonfunctional requirements different from traditional applications orientated to a personal computer. Some of them are the optimization of the resources related with the battery and the access to remote data, and a different design in order to have a good level of usability.

4.1.2 Development of the Back-end (API)

As we have mentioned before, this service is fundamental for our system this service in order to be able to share the lists. In the creation of this API there is not any complication, because if it was only a practice for a subject, this would be implemented and then would end up in a code repository. But, if we want to develop good software, we will have to prepare the API for a possible versioning, the exposition to Internet, adaptability to the changes and other problems that during the development we will have to face. Consequently, the first goal is to develop an API easy to maintain and be aware of the possible changes that might appear in the future. From the changes in the business logic, such as the fact that a client might have more than one email, to the integration with services of third parties, distribution of the databases and others. We will drill what we think that is essential:

- It has to be functional and carry out the processes outlined in the business logic.

- It will have to follow, if it is possible the REST standards for the API.

- It will have to be ready for versioning

- It will include a first authentication version both in the consumer level of the API (mobile app, web) and user level (which user is making the request).

4.1.3 Development of the *Android* app

In order to access to the information that there is in the cloud an application will be created to the easy interaction of the user. In order to facilitate this use we will try to follow the conventions and recommendations of design to make an interface more usable and friendly.

- It will have to be functional and carry out the most basic actions set in the business logic.
- It will have to follow the standards to make a more usable design
- It will have to be prepared for versioning, having a control of the versions that actually are supported for the API and will obligate, if it is necessary, the client to update the application.

Although it is the most visible part and the one that the final user could appreciate, we will leave the esthetic design in the background, because the estimated workload for the other goals is high enough. Consequently, the application will be defined as a prototype and we will not allocate any resource to its esthetic. If we will not have enough resources or time we are thinking in not implement the application and only make the main design of the screens and think in the architecture design.

It has to be said that with the first research we made about best practices in Android, we did not found too much information and what we found did not help so much. As the first goal is to work following best practices and applying a correct design, we prefer to dedicate time to think carefully and leave the implementation for the future.

4.2 Scope

The realization of this project was planned in 5 stages.

The first has been intended to read papers, articles and documentation about technologies to have a first contact with them. Install a virtual machine, tests with frameworks, configuration of the test environment, etc.

The second stage has been intended to write the documentation referred to the scope, the state of art, planning and budget, among others. It belongs to the subject of GEP.

The third stage has been intended to a first implementation of the API's logic. We evaluate to develop simultaneously the API and the application but, because of a lack of knowledge and the will to establish more bases, we opted to implement first the management of the users, lists and operations of insertion, modification, etc on these items, before to continue. In this stage is where we will start to apply in a practical way all the best practices that we want to follow and where we will dedicate a lot of time in making a correct design of the app.

During the fourth stage is where we wanted to develop the application because in this stage it could be integrated with the API and the functionalities of messaging like the sending of emails or the notifications that the users would receive in their Android devices. Moreover, we wanted to implement all these functionalities in the part of the API that we could consider that are necessary or gave a significant value.

The last stage we wanted to dedicate to the upgrades and new features that may have arisen during the project. In the case it has been an important deviation of the time because of problems with technologies, or because a bad foresight of the workload, an elevated cost in time terms to make a good design or, simply, or other complications, we wanted to dedicate this stage to finish the development of the others parts and expand the coverage of the test in case we believe that it was insufficient. This has been the case and we have used this time to continue with the development and to write the documentation, more extensive than we previously thought.

4.3 Technical competences

In the inscription of this project we chose some technical competences that we will justify in the following lines:

CES1.1: Develop, maintain, and evaluate systems and complex and/or critical software services

Grade: **Enough**

In this case this competency is mainly chosen for the part of development. In this project we have developed a system from the beginning, and we have taken into considerations the future maintenance. Although what it is shown in this project is a small version, our intention is to continue with this project and make it grow with time, using it as a practical part to be able to apply everything we have learnt. Also, as we have mentioned, our intention is to develop a system that is easy to maintain, because in the future will be ourselves the people in charge to do it. Consequently, although in the project there is not any system maintenance, we have thought, orientated and evaluated the designs to make easier and simpler the maintenance. We consider that we have accomplished perfectly this competence.

CES1.2: Give a solution to the problems of integration depending on the strategy, the standards and the technologies available.

Grade: **A little bit**

Our initial intention was integrate the app with systems of other supermarkets in order to take their products and be able to show them in the app. After sending some emails, we realized that this was not possible. We neither have accomplished to integrate with the application in a practical way because we did not develop. However, in both cases we have considered the situation and we have thought some designs to do it and decoupling in the best possible ways from these services, which belongs to third parties. We consider that we have accomplished this competence in a theoretical frame but it is needed a practical application.

CES1.4: Develop, maintain and evaluate services and applications distributed with the support of the network

Grade: **In depth**

This has been our fundamental pillar: think a system which was distributed, with all the problems that it has. Although one of the parts is not finished, because of the over-costs in time derived from the concern about best practices and correct designs, we consider that we have accomplished this competence in a high grade. The fact of having into consideration all the problems derived from the architecture distributed, such as design and evaluate it is why we think we have accomplished this competence. Currently, there is only one of the two parts, but it is the base in order to other clients could connect. If we would want to change the client and instead of using Android we wanted to use a web Single Page Application it would be possible because our design allows it.

CES1.5: Specify, design, implement and evaluate databases

Grade: **Enough**

This was our first concert, even before of concerning about best practices. When we began to design the system we evaluated which would be the main problems we will have to deal in the database and how we could solve them. The first (and most important) was the fact that in the table of the database which initially save products could grow excessively and maybe could "overflow". We thought different ways to solve it but, after thinking a lot, we decided to leave this problem on a side and deal with it when it appears. This way, the hexagonal architecture gave us the solution: divide the access to the database in repositories and inject the implementation that we were using. All of it without having to modify the business logic.

After talking with some experienced people in database, they said to us that to achieve saturation in the table of products, the application must to grow a lot.

Moreover, the persistence of our system is made on a relational database and, consequently, we consider that we have accomplished this competence in a higher grade.

CES1.7: Control the quality and design tests in the software production

Grade: **Enough**

This is a competence we have worked permanently throughout the entire project. Our way to work (TDD) has obligated us to implement test for all and each one of the functionalities. Consequently, besides of design them we have implemented, and when we were developing new functionalities we detected little errors in our tests, which we have solved and improved in the next tests. We consider we have accomplished this competence and even we have interiorized this knowledge.

CES2.1:Define and manage the requirements of a software system.

Grade: **A little bit**

Although it was not our initial intention (dedicate to write these requirements), we have been forced to do it. Moreover, we have worked in alternatives ways of writing requirements different of what we have learnt at university. At the end, we have written the requirements through users histories. Also we have written the nonfunctional implicit requirements implicit in the development such as orientate the system to an easy maintenance, a code with test or a good design.

We consider we have accomplished this competence in a practical way in a grade higher than initially expected.

CES2.2: Design appropriate solutions in one or more application domains using engineering software methods which integrate ethical, social, legal and economic aspects

Grade: **A little bit**

This project answers to the need of being able to manage shopping lists in a shared way. We could have offered the solution of sharing a Google drive but this has not been a final degree project. Our initial idea, previously to consider this project as our final degree project, was only to respond to this problem and learn to apply best practices in software designing through a practice exercise. Although we have not studied ethical, legal or economic implications of the project, we have already worked on the social aspect because this project intends to improve a usual process in our environment which is the fact of making a shopping list.

5 Methodological and technical thoughts

In order to guarantee the application of these best practices and a correct organization we have opted for following different methodologies, some more specific to the development and others closer to management and organization.

5.1 *Test Driven Development*

One of our goals within the application of best practices is the use of *Test Driven Development*. This technique proposes a development cycle by functionality, as we could see in image 3.

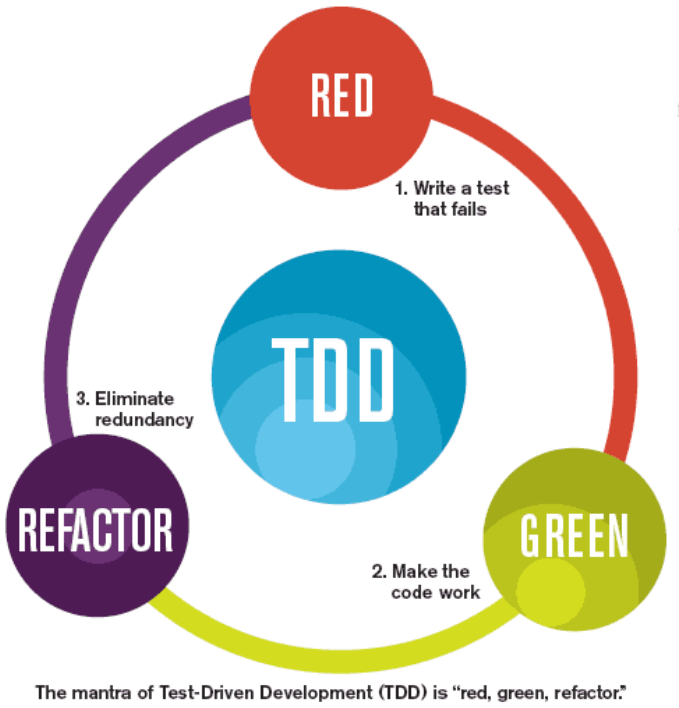

Figure 3: TDD Cycle

The idea would be to choose a functionality (total or partial) to implement and think on which tests would validate its behaviour. For example, recover a user of the database. Before create the tables and the classes that will make these, we have to consider which want to be its behaviour in each case: if we will pass and ID, an email or both; if will returns an exception or a null value; if will return an array with fields or a domain model. Once we know what we want to does this functionality even without writing code, we write the functional tests and run it. Obviously the tests will fail, we have not wrote anything yet. Here begins the simple but difficult task of write the right and necessary code to pass the tests, later we will see why. Once we have the code what pass the tests, we can if we believe it is necessary for design reasons, we can refactor with the absolutely security we are not changing the behaviour of the feature. If the features change, and tests are correctly written, tests should fail showing that the behaviours has changed during our refactor.

This process is repeated for each one of the functionalities to implement. It is not an easy process to internalize and even less easy in the way we have learnt to work: firstly we write the code and then we test it, but this helps us to consider what is what we expect of what we want to implement. However, is a process that contributes with some advantages:

- Software without bugs.
- Facilitates the trust in the work of other people. If everybody writes its test properly, modify the code and suddenly, the tests stop working, we can deduce which is the part of our code that provokes the error. If not, once finished the functionality (without tests), if the software does not work we could not determine if it is the code we have just written or a code written in a previous moment.
- We write the minimum code needed to solve the functionality and avoid useless code.
- In the long run we will be more productive. It is easier to localize the error because in the test, if it is well written, indicates what it is testing and reduce a lot the time of searching of bugs.
- Allows us to detect possible cases not contemplates before. In the case that the functionality specification is given, if we are critics writing test, we can detect possible cases we have not contemplate.
- Allows the introduction of junior developers in a safer team. If we want that a inexperienced person implement some little functionality but we cannot control him or her to check his/her work, we could leave test and simple if the code written by this junior developer pass the tests, we can trust in his/her code and work.
- It allows to make refactor safer.

Seen in this way it seems very beautiful but the truth is that develop a code that way could bring a lot of troubles. If you have to make an implementation using code of a third person, classes that are not implemented or external services, applying development guided by test could seem impossible but it is not: in the next section we will analyze.

5.2 *Unit testing* and *Mocking*

When we talk about applying TDD and work in large teams or depending of external services questions arises. How I run the test if I do not have a class which depend? How I request to the extern service an image if the request is filtrated by IP? Very simple, freeing ourselves of these dependencies. At this point, we arrived to the reason why we have applied TDD and unitary test to our project: it allows detecting bad designs.

One of the common mistakes in implementing tests is the following one: This pyramid

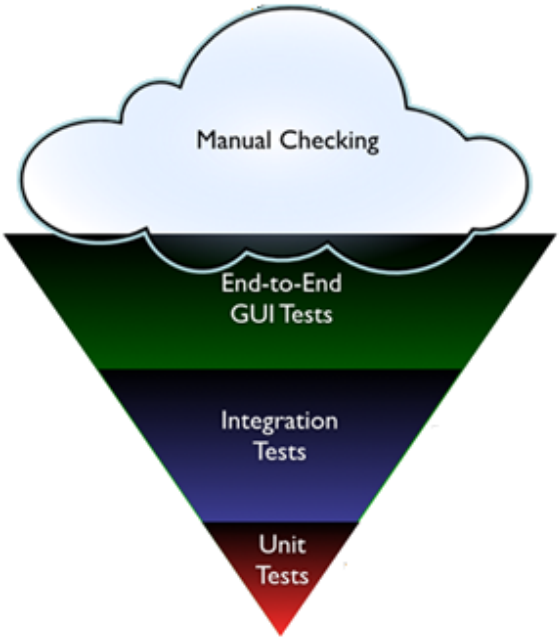

Figure 4: Software testing Ice-cream Cone

is known as the Software testing Ice-cream Cone[7], a bad test implementation according to some opinions. It is true if we have this in our project we have more than anyone who have not got any test and this is, at least a small best practice but we are far away from where we want to achieve. Although this way of writing the test could give us some advantages, as we mention in the TDD section of the project, does not offer the advantages that TDD and unitary test has: detect design error. With integration test we usually check the functioning of more than one cape and the interaction between them but does not allow to check if the integration or the way of using other parts is made properly.

If during the implementation of unitary test is very difficult or impossible to write, the tests itself indicates we do not have a good design. One of the best practices in the software design is the injection of dependencies, which allow us this independence between classes. For example:

```
public class ServiceMailSender {

    public void send(String message) {
        MailSender mail = new FrameworkMailSender();
        mail.send(message);
    }
}
```

If we want unitary test on this function we found a problem (besides coupling with framework). We cannot independent ourselves when you make the tests about who send the emails. Here we could detect an important error of design. What will happen the day I change the framework?

One of the solutions which allow solving these problems would be, as we have mention apply the injection dependency.

```
public class ServiceMailSender {

    private FrameworkMailSender sender;

    public ServiceMailSender(FrameworkMailSender mailSender) {
        this.sender = mailSender;
    }

    public void send(String message) {
        sender.send(message)
    }
}
```

Although this situation allows us to extend class *FrameworkMailSender*, pass to an instance which would overwrite the *send(String message)* and implement a unitary test which checks if the mail is sent through a *stub* [8], we see that there are things that does not have good look, like as having to extend a framework class when, precisely we want to release of framework and specific implementations.

The proper solution (summarizing) and the one we have implemented in a similar problem to our project is the following one:

```
public interface MailSender {
    public void send(String message);
}


public class FrameworkMailSender implements MailSender {
    public void send(String message) {
        MailSender mail = new FrameworkMailSender();
        mail.send(message);
    }
}


public class ServiceMailSender() {

    private MailSender sender;

    public ServiceMailSender(MailSender mailSender) {
        this.sender = mailSender;
    }

    public void send(String message) {
        sender.send(message)
    }
}
```

This allows us to inject the service to any kind of class which implements the interface *MailSender*. This allows us to inject the service to any kind of class which implements the interface

Once we have created the design, we realize that make a unitary test of this service is easier than expected. Here we have two options: or use a test framework which facilitates the work or implement manually the stubs and mocks. These objects allow us to check if, truly, the class *ServiceMailSender* works as expected. They also allows to check if on an object have been created a method, how many times have been done and what we want it makes as an answer.

Basing on this last solution, the test would be the following one:

```
String message = "Hello";
MockMailSender mock = mockThisClass('MailSender');
mock.shouldReceive('send').oneTime().with("Hello");

ServiceMailSender ms = new ServiceMailSender(mock);
ms.send(message);
```

This code lines allow checking, without send a email, if the class *ServiceMailSender* have the behaviour we expect. To more information about mocks we recommend the reading of an *IBM* article [9].

Once we have seen what implies to make TDD and unitary tests using (or not) techniques of mocking we want to emphasize the advantages to apply all this work:

- High quality software when applying the necessary patterns to facilitate the structure of the tests.
- Application of SOLID principles, or at least, the detection of the violation of these principles. Fix them depends on the developer.

Emphasize unitary test allows us to fix almost all the problems that could come up when you develop using TDD. It is more efficient to use TDD with unitary tests.

5.3 *Scrum*

We have considered developing the project following the agile methodology *scrum*, not literally but a small adaptation. Every scope was subdivided in *sprints* in which we tried to finish the functionalities, always giving priority to those which were critical or those which our clients emphasize. We consider critical those functionalities which were needed to develop other functionalities.

Each sprint had to include an analysis of the functionality, the design, the development and also the testing; trying to obtain by this way functionalities always finished and prepared to deliver. We wanted to assure of this possibility of delivery generating the test at the same time as the code.

The adaptation of scrum we wanted to do consisted in develop lately the user interface and not simultaneously with the server. In a standard scrum methodology we would have to develop, for example, the user authentication to all levels in an only sprint: user interface, API connection from the application and logic authentication to the server. By this way we obtain a finished and deliverable functionality. However, due to several reasons, which we have already mentioned, we have decided to develop first the logical part of the server, forcing ourselves to make an adaptation of our needs of the scrum methodology.

But the reality have been different.

We wanted all the functionalities were proposed by people close to us and also were potential users. Although initially this people showed their will to make proposals and imply themselves in the project, when they realized the time that they had to spend to specify a card or functionality, they desisted. Also there was the problem they was not people familiarized with writing user histories, requests, or even to clarify the ideas about the application they wanted to transmit. We were not able to find anybody capable of making the exercise of product owner and dedicated some time to think functionalities, write them down and make priorities among them. Consequently this role was assumed for us. But working with scrum also implies to use a board where write the tasks and then the product owner establish priorities among them. This implied to have to maintain the board updated, in addition to think about the application design. Finally we dismiss this option and the board was abandoned.

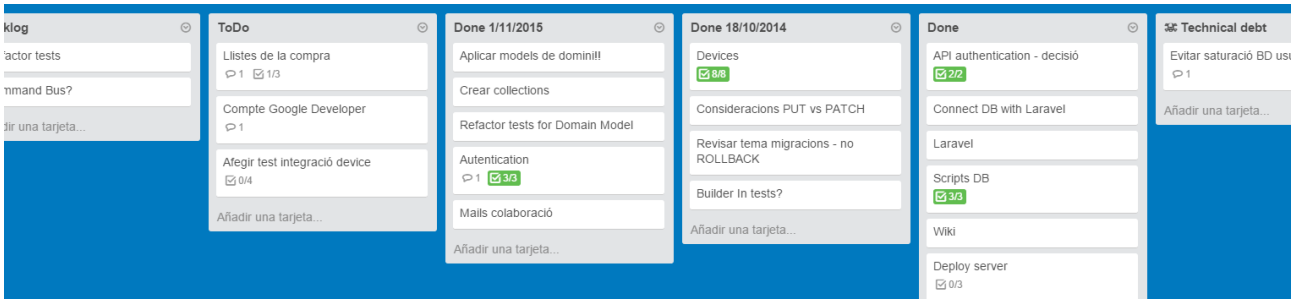

Figure 5: Outdated Trello board

6 Tools

6.1 Management

6.1.1 *Trello*

As we have already mentioned in the scrum methodology is necessary to have a board for organize the tasks. *Trello* is the tool we chose to manage this board. It is thought to use in any agile methodology and, as it is free and very complete we opted for use it.

We made a separation of the tasks in four groups:

- *Backlog*: in this group there were prioritized tasks which were pending to be done through the project.
- *To Do*: this were the conjunct of tasks that had been schedule during the current sprint.
- *Doing* or in process: there were tasks which were doing at that time.
- *Done* or finished: tasks already done and validated.

We ended to leave this part because it consumed a lot of our time and we believe it did not contribute a lot to the project, because we spend too much time creating cards and specifying the tasks to make and managing all the board.

6.1.2 *Microsfot Project*

We began to use this tool in GEP when we had to do the planning. At this time we believed it was a complete too that allowed to manage perfectly the project.

As a recommendation of the GEP's teacher, we started to bring a count of the hours we dedicate to the project and, by this way see how evolves the project and which deviations it suffered. Initially, we believed it was a good idea because despite we have to spend some time in the management of the document, the result worth the effort.

Throughout the project, as days passed and we were adding dedicate hours to the project, we realized the deadline was closer and we have little code. It was a tool that demotivate a little bit to keep making things properly, because we have very well write code but a little code. For avoid to fall in a pessimism spiral we decided to stop using it because we did not want that rush weights more than job well done.

6.2 Validació

First of all, before talking about validation tools by themselves, we would rather introduce the way we have projected the validation in this paper. We have projected it in two different ways/focus; one more functional and focused on the business processes and the other one focused on guaranteeing the control of best practices and an adequate design of the software architecture.

Focusing on the first one, this validation has different three parts. The first one, iterative within each functionality, throughout the tests. We have been validating that the code run as expected and every single function was free of errors.

We had foreseen the second initially at the end of each sprint. However, as mentioned before, we have ended up working by functionalities rather than codes. So this second step was given at the end of each new functionality at getting integrated with the others. In this new step we have checked that the system adds works properly and in most cases have been a mere formality, as we have followed TDD in a strict way, everything we added was free of errors. This second step has been achieved at first try satisfactorily in our project. The fact of having this test battery has allowed us to guarantee that every time we added a new code it did not break the previous content.

Finally, we had considered a validation in the final delivery in which it was feasible to carry out the business processes defined in the requisites. Nevertheless this new step has not been overcome in terms of implementation levels.

The second focus, more addressed to the correct design of the software and the control of the best practices has been harder to evaluate in a theoretical way, and at this spot is where the practice and helped us.

Initially, even that we were aware what was right and what was not, at the time of implementing it was hard for us to recognize at first hand which were good designs and which had flaws. It is because of that that the first analyses and the first decision-taking were tough. We were aware that we would have to take these decisions through the entire project, and that in case of mistaking, the time needed to do refactor would be way more than the time we had.

One of those questions was, for instance, whether using the models logically or not. After some days thinking and debating with colleagues we reached the concept of anemic model and rich model, where the former is a data model and the latter has logic. Carrying a research about anemic models we saw an article from Martin Fowler in which the anemic model[10] was described as an anti-pattern. Once we read this article we decided to try on developing towards adding logic to our models. After a couple tries through the development, we saw that not on out designs, the model was still anemic. This situation worried us, as despite of being aware that this was considered a bad best practice out solutions aimed here given that some solutions more fitting to the rich models were quite hard to implement and did not seem neither easy to sustain not being tolerant to changes. After further reading regarding this subject we found a couple posts that talked about CRUD and REST[11][12] and how these favoured anemic models. Finally we opted for anemic models, as given the functionalities that we ought to implement; it seemed easier with the anemic model. Similar to this decision there were others that required some time from us to achieve the most adequate design. Now that we can look back and see the reasoning and the solution with some distance, we think we chose wisely.

Despite the readings, the debates with colleagues and the valuation of all these best practices, we realized that there is no better validation than keep developing code. Some of the designs initially chosen were modified at the time of developing new functionalities, as they did not allow to keep developing comfortably or we observed that they were against some of the best practices like principally getting independent from the framework. We think that if we kept developing code we would find details in our design or little refactors, however, we think that the basis we have established are good enough to avoid having developing problems regarding technical debt or bad decisions.

As we have previously commented, we were helped and we were doing a good design using the TDD and unitary tests. If the test implementation was easy, it was not excessively hard to think and were quite complete, it indicated us that until that moment our tests were quite good. Up to this point

Along the development we have been acquiring experience and fluency at the time of recognizing our errors and that has permitted us at the end to develop much faster than at the beginning.

The tools that we have used for the validation have been *PHPUnit* [13] and *Mockery* [14]

PHPUnit is a tool for creation of unitary tests that has allowed us to check that the developed functionalities worked correctly and in second time jointly with the TDD, as we have explained before, detecting design errors in our implementations. This library of test uses assertions to check whether the code functionality is the expected one. The associations allow us to check behaviours like if the result of a function in null, empty or equal to a variable among other cases.

The mockery library has eased the work a lot at the time of writing tests. It has permitted us to create mock objects easily and checking the state of those with simple calls- The mocks has principally make easier a tailored interaction with the dependencies of one kind. It has allowed us to create an object with a series of specific answers to a previously defined interactions. It has been the last piece, small but important, that has allowed us to develop with TDD the unitary proves of our system with relative ease.

6.3 Development

This project has been developed in *Linux* and *Windows*. settings. The software development of the back-end has been done in a virtual machine *VMWare* in which we have installed an *Ubuntu 14.04 LTS Server*. We had the intention of using the same operating system in the pre-production server. In the Windows setting the *Android* app development will be carried out.

The tools that we have used in this project are the following:

- *VMWare*: virtualization software that has allowed us to use a virtual machine with the aforementioned *Ubuntu* version
- *PhpStorm*: *IDE* from *Jetbrains* for the development of *PHP* code. We have opted for this tool because it offered us some helps as for the refactor or the integration with the *PHPUnit* that was of great utility. Now that we have used it we consider it the best *IDE* we have worked with until now.
- *AndroidStudio*: *IDE* from *Jetbrains* for *Android* apps development. We have not explored it as in depth as with the *PhpStorm* and we lacked usage. Nevertheless, we regard it a much better alternative than Eclipse to *Android* development.
- *Git*: tool for managing the control of versions and protecting the code from possible problems like data loss. It has eased our development the fact of having the control of the evolution of the code and it has permitted to see what we have modified since the last commit and go back when the tests failed.
- *Bitbucket*: this platform allowed us to access a central code database from anywhere we wanted. Sometimes we needed to develop from a laptop for locational reasons and the fact of having code from the cloud was really useful
- *Laravel*: framework that we chose for the framework part. This tool eases the access to the database, the processing of the *HTTP* demands, the injection of dependencies and so. It eased a lot the wide documentation it contains: one summarized with examples of how to use it and one highly detailed documentation of the own classes of the framework.

- *PHPUnit*: in spite of having already mentioned it in the previous section as a validation tool, we would like to highlight its functionality as an easing development tool
- Pre-production machine: Machine located in the cloud and rented from the platform *DigitalOcean ™ Inc.*. with the following features:
  - Cost: 5$ per month
  - RAM: 512 MB
  - Processor: 1 core
  - Disk: 20 GB SSD disk
  - Transfer: 1TB

  Despite we could not deploy in this machine initially, in a first read about deploying and server maintenance, we initiated a step consisting in the implementation of security measures. These measures were as diverse as follows:
  - Creating new users without administration permit to access the server
  - Disabling the administration user
  - Enable the automatic updates
  - Install control and monitoring tools
  - Configuring SWAP
  - Checking the *openSSL* version in order to avoid versions with bugs

  And many more actions that we were compiling and started documenting in the *wiki*. Due to the time that it requires and what provided to our project we decided to leave it for later. Finally this task was abandoned in order to focus our efforts to the development.

7 Temporal Management

In this section we will exhibit the initial planning done, how was temporally developed the project and in which tasks we invested time. We will analyze the deviations occurred and try to find the reason they happened.

7.1 Initial Planning

All the tasks and its temporal organization regarding the initial planning of the project can be seen in the chart at the attached Appendix A A and its correspondent Gantt Diagram at Appendix C.

Initially, at the GEP subject we carried out a project planning trying as accurate as possible to describe the different steps and the length of those. This planning was exhibited in this section. It needs to be mentioned that is a reflection of the tasks that we considered that need to be done and the possible challenged that we could find.

At the chart 1 we can find a summary of these tasks.

Name	**Start**	**End**	**Total hours**
Preparation and evaluation	14/06/2015	14/07/2015	136 hrs
Development (API)	01/09/2015	27/10/2015	156 hrs
Development (App)	02/11/2015	16/12/2015	132 hrs
Final delivery and documentation	17/12/2015	11/01/2016	72 hrs

Table 1: Planification stages

7.1.1 Preparation and evaluation

Before carrying out any planning or starting the development, previously we performed some work in order to obtain some basis that we considered indispensable. We could fin the following tasks within this basis:

- Creating a virtual machine as similar as the one hired from *Digital Ocean*.
- Choosing a framework taking into account that it could offer a wide range of told that eases certain extern functionalities to our business logic
- Installing the development tools, in this case, *PhpStorm*.

- Installing the framework and its dependencies: *MySQL*, *PHP* and *Apache*.
- Creating a code repository in *Bitbucket*.
- Reading about how to increment the security level in a server
- Creating a *wiki* with the definite steps on how to recreate the environment
- Decision making about:
  - API versioning.
  - Authentication in the server both in the app and user level.
  - Project architecture
  - Reading about testing
  - Best practices to follow

We considered all these tasks as part of "sprint 0" in which the technologies necessary for the development are needed

7.1.2 Initial Planning (GEP)

The weeks from 14 September to 18 October of 2015 were assigned to the GEP subject in which we did this same planning and another project documentation

7.1.3 Development

This stage corresponded to the creation of the deliverable product. Here we should produce the incremental iterations of development, where the implemented functionalities would be validated and the quality checks will be done with the client. At this point in the planning process we did not consider that there might be no clients who could dedicate their time to think about the functionalities and review them with us. Apart from these validations with clients we also included validation through test, a much more technical focus and remote from business.

We took two weeks as a standard length measure of a sprint unless otherwise indicated.

Sprint 1:In this first sprint we had to invest the time to create all the creating and updating logic for users on the server level. We thought that the functionalities would be easy and we could devote most of the time to improve the knowledge of the framework, to apply architecture patterns chosen and to test the TDD[5] work methodology .

Sprint 2: : We wanted to devote this second work stage to implementing the user authentication. We had the intention to evaluate different options, document them and decide which one best fitted our needs. We wanted to add the related part regarding mobile devices: charts in databases and the creation of *Google*'s developer account. We wanted to contact hypermarkets and supermarkets in order to know whether they could or would collaborate with the project or not.

Sprint 3: In the third sprint we had to allocate all the resources to design and program everything related to shared lists. From creation, modification, deleting, and most complicated, simultaneous editions. This sprint was already expected to be complex and lengthy for which we estimated 4 weeks. It was a sprint excessively long. If this length confirmed we expected to split it into two parts: design and lists programming without considering the simultaneous editions and the second part of these editions that we expected would be problematic

Sprint 4: We reserved this week sprint for possible problems with the API, both with ill-defined functionality, as not passing validations or others. Once all the validations had passed we will proceed to the deployment at a pre-production server.

Sprint 5: At this stage had to do the resources deploying resources in order to start the application developing. Installing *Android Studio*'s *IDE* and drivers. We wanted to start introducing the first tests with the *JUnit* tool. This adaptation sprint was expected to last only a week.

Sprint 6: It was expected to develop the flow of user's creation, the creation of a list and the first integration with the server part.

Sprint 7:In this sprint we had the intention tom work the more complex part, which was the addition, modification and deletion of items from a shopping list. As we anticipate that it would be a complicated stage we estimate that it could last about three weeks more or less.

Sprint 8: In this final sprint we wanted to close any topic that could have been left half-done and validate that all targets were met. However, we were aware that there could be deviations, so we had it reserve to fix potential bugs.

7.1.4 Entrega Final i documentació

With all targets met and validated we had the intention to perform a release at the API's pre-production. Concerning the app, we wanted to release it in the *PlayStore* as a *beta* stage application, in order to try it with some volunteers.

7.2 Course of the projects tasks

All the tasks and their temporal organization referring to the project development could be seen at the chart in the attached appendix B and its correspondent Gantt diagram at Appendix D

In this section we will see which the real development of the project was. It needs to be mentioned that the accuracy of the tasks in the last two and a half months (end of November, December and January) was not as accurate as possible. As we have mentioned before we reached a point in which carrying out this control discouraged us to keep working in the project and in order to avoid this distasting sensation we decided to assume that there was a delay in the planning of the project but that this delay was due to the work with the objective of building quality software.

The stages of *Preparation and evaluation* and *Initial Planning (GEP)* did not vary in comparison with the original planning, given that this planning was written once these stages were developed. In spite of this, the *Preparation and evaluation* stage has suffered a delay along the whole project. We considered that in this stage we would take all the decisions and once we started developing we would not need to invest more time in this task, but it has not been like this. We did not think about many of the situations until we step across them and needed to read about them and take a decision right at that time.

At the end, the stages have been distributed more or less like this:

Name	**Start**	**End**	**Total hours**
Preparation and evaluation	14/06/2015	14/07/2015	136 hrs
Development (API)	01/09/2015	06/01/2015	292 hrs
Design (App)	14/12/2015	24/12/2015	27 hrs
Final delivery and documentation	09/01/2015	18/01/2015	42 hrs

Table 2: Actual development of the project tasks

7.2.1 Preparation and evaluation

This stage has been lengthier that expected. On one hand there has been a series of tasks that once finished has not been needed to review as for example the environment creation, the framework selection or the *wiki* redaction. However, everything referred to the decision-taking process has been lengthened across the whole project transversely. As much as we could read about rich or anemic models initially, we came to understand what offered each of them and opt for a design: was not until we implemented it that a lot of doubts appeared that we need to solve right at that time, and we had to read and document ourselves more about the subject and ending up taking a decision

7.2.2 *Back-end* (API)

As the development of the project was not developed in sprints as we had considered initially, we would divide it by functionalities that is how we have worked and in that way we could detail what have been developed in each functionality, which problems arose and what we have considered to fix it.

User creation

Fully developed functionality. Already in the first one we found that we did not have product owner and we had to design what the application would have, would send and what was expected to be received. This was the first problem, given that a task that we initially considered only designing the architecture and developing, we found ourselves that we had to think how the creation screen would be, what it would send and what would it need as a response.

Furthermore more technical doubts appeared as we could not apply rich models and we had a design with anemic modes, which answers need HTTP to return, if we wanted to answer like a public API or if it was worth developing the *Command Bus* pattern. Document ourselves more, value and make a decision implied some time that we not expect firsthand. What are more we though about an answer on how to avoid an attack against the server in case someone wanted it to collapse creating an infinite number of users. At the end, no solution appeared to us an effective enough and we choose to leave it for later and control it in another way rather that through the business logic.

It needs to be said that this functionality has suffered quite some refactors as more we learned more details we could observe. There are some decision that still now we could not know if are the most suitable. Going on with the developing will answer this.

User updating

Initially we thought that this functionality would require less time given that we would adapt to the technology and among all, no doubts would arise. But it has not been like this: we had to value whether and update would be idempotent [15], whether implementing a full or partial update, which field could not been updated and which answers HTTP give in any case. As we developed we could see that there were things that we did not perform correctly and we performed refactor. Here, the previous tests were really helpful, and things that even being correct could be widely improved. In fact this sensation has been occurring along the length of the project repeatedly.

Choosing and implementing the authentication method

This was a subject that concerned us; we had to guarantee that the information of a user could not be accessible to the rest of users. We considered different methods but we need to take into consideration that we comply with the REST standards that literally say "communication must be stateless""[16]. So we wondered between two options: an authentication based on tokens with a temporal time-life if not used in a while, or implementing *OAuth*[17]. Later we will explain why we decided to choose the token authentication. Moreover, new security problems arose again.

Once decided the method, the implementation brought some problems when we wanted to get independent from the framework. After discussing it with different colleagues we opted to follow the following focus: using the layer of *middleware* that the framework offered us (we use it because of that) using an authentication service. Me maintained some independency with the framework given that this service was responsible for knowing whether the token of a user was valid or not, and the only thing that the *middleware* layer could do was to capture the request, check if it has the token and as the service whether the authentication was valid or not. In the case of changing the framework, we would only need a couple lines that perform the same adapting ourselves to the language or method of the framework in order to deal with the HTTP requests.

Preparation for devices

We considered that the API would be used from a mobile device and that this could bring an advantage: avoid push notifications. This would allow us to notify all the devices when there was an update ready in a list and minimize the numbers of requests to the server in order to know the updates.

In order to carry out this, we needed to inform ourselves and read about how the *Google Cloud Messaging* servers work. It was not a costly task; in fact it was the easiest we performed until that moment. The TDD and the unitary tests allowed us to implement this functionality without owning a *Google developer* account. We were sure that what we had implemented worked but we still needed to verify it with the integration tests. Having a great number of unitary tests is perfect, but if there are no integration ones, we do not have the certification that everything works according to the tests. This statement might sound weird but the idea is that the unitary tests could verify us that the class behaves as we expect. We only needed to check that we were interacting with the API of *Google Cloud Messaging* as expected.

Sending mails to propose collaboration

We sent the email to different hypermarkets in order to ask for collaboration. The proposition of collaboration was easy: integrate us with an API of them to collect their products and integrate them in our application. All the replies were negative, some of the replies were sent without reading the proposition such as *"we are not currently interested in the development of new apps"*. Others were declined as not being able to offer access to their API and the remaining commented that they already had other projects on the way and could not support our request. Summing up, we did not have access to an extern API.

Creating the Google Developer account

This task and its apparent easiness needed more time that what was expected. It was not the account creation itself that require efforts and time, it was the fact of planning and implementing the integration tests what required the volume of time to increment. Once we had the account we had to decide how to add the API-key to the project correctly. It is obvious that we could have added it as a string in the middle of the code but it is clear that it is a best practice. We needed to develop a class that allowed us to read configuration files getting independent of the framework and prove that everything integrates correctly. Luckily, once developed, the tests passed at the first attempt.

Once finished this task we started to read about best practices in *Android* and we started the design of the application development. For that reason we will discuss it in the part of the application development.

Design of the parts of the shared lists

In the planning we made an important mistake: we did not think in all the implications of this task. The design of a model, how would it be displayed in a database, which attributes may it have and which business needs would it attend to would have implied the 12 hours that we initially estimate. In spite of this, we did not think in all the cases that the fact that two users may be interacting at the same time over the same list and same products may imply. When we consider this problem we could not abstract of who would use the API that at the end were mobile devices. We should reduce the number of connections and data transmitted as much as possible.

This task took the next two weeks without being done, as we could not find a proper solution between solving the simultaneous edition or minimizing the number of requests.

Implementation of the creation and modification

At that point, we realized a poor implementation. Instead of returning our own collections, we returned arrays. This blocked us from applying certain actions or turning them into strings as they were objects upon which we could not apply methods.

Furthermore, we had some troubles that we did not know how to face regarding the repositories. If we ask for a list at *ShoppingListRepository* and this returned which users belonged there, these should be identifiers or user models? If we opted for the former, we had to find the users with those identifiers and obtain their information. If we opted for the latter we obtained all the information from the database in just one demand. Later, we will see which option did we choose and why. Just mention that there were obstacles due to the design.

The last impediment we found in this task, already mentioned, was the poor specification that we did. Within the concept of list there is an inside concept that is product. We did not think which would be the concept of product in our application and some other problems related with the products themselves.

Implementation of the case of the simultaneous edition

Watching in perspective the planning that we did we notice a mistake. When we planned the task, we did not solve some and we perform an incorrect estimate due to our not-so-precise vision on the implications of each case. If we had solved the simultaneous edition task we could have seen that there were many decisions to take into consideration and take that would imply much more time than estimated. Here we made a mistake.

This last functionality has not been solved successfully because we have opted for a solution that would not solve the simultaneous edition from a business perspective. The current implementation would overwrite the last item; the last edition is the valid one.

Revision and testing of the API – Deploy at pre-production

We contemplated the revision and testing stage as a step to expand the coverage and revision of potential mistakes or as a contention period in case we did not finish the API having a margin of time. We realized that if the task is done correctly, not in the sense of correct developing, but in the sense of thinking about the tests correctly and implementing them with care, the chances of making mistakes or the need to add more tests would be really low. At the end, we did not arrive to this stage, we had to invest time in previous functionalities.

7.2.3 *Android* app

Most of our resources have been used during the project in order to develop the server part. However, during the push notifications sending development we wanted to know if the correct integrations of both parts would work: sending and receiving. In order to carry out these checks we needed to introduce ourselves in the *Android* development and do some reading about best-practices.

After reading some articles that did not convince us, we asked to acquaintances that worked with Android which best-practices did they follow and which recommendations they could make. The answer was discouraging: nobody could mentor as exactly in this aspect. They could in technical stuff and how to develop functionalities, but not in how to structure code, unitary tests or injection dependency.

We chose to implement that part, following the best possible practices that we believed and devote the remaining efforts to finish the back-end part. We decided to finish an API well done and well-structured rather than a half-baked API and an application without any best practice included or correct design, We considered that if we wanted to implement the app successfully we should devote much more efforts and resources that we could possibly devote at that time and we neglected it.

and we neglected it. The only part that is currently finished is the register of the device at *Google Cloud Messaging* and the reception of notifications.

We lacked a lot of Android technical background and even more in good-practices. Is a topic that could fill a full final degree project.

7.2.4 Final delivery and documentation

We could not devote as much time as we would have liked to this stage. Due to the poor initial planning, we did not have the expected time and we had to cut back resources in this stage, despite being of dire importance

7.3 Deviation analysis

The development of the project has been influenced by a lack of knowledge in best practices in the development and, also in a relevant way, by the lack of internalization of these best practices. Although knowing what is testing, what contributes to the project, how to make it and all the advantages that it offers, at the beginning it was too costly to try making TDD. The most repeated question was: what I write in the test and by which I start? We did not have training so it was difficult to develop on a good rhythm. But with the pass of the time, cost reduced and this important overhead disappeared over the time. This does not imply that making testing was an automatic thing and do not imply time, but the cost was reasonable.

The other problem has been the exercise of different roles. Initially we thought a project where somebody close to us would give us ideas, what he/she expect and we could talk with him/her about functionalities. But this person has not existed. We did not think this person must to be somebody who knows what implies *scrum*, a person close to the business or app we wanted to develop and a person who spend some time in the app. Finally, we have had to assume this role, consider the functionalities, design the flow of the actions, think about the user interface, what could benefit the final users... Assume this role has been an unexpected extra time. We did not think that in a project where we had to apply *scrum*, the product owner must be ourselves. Moreover, we have also to assume the role of architect and developer. And, finally, also be careful to apply the best practices and methodologies for the project, regardless the role we were assuming.

Another problem, different from the other ones, has been a too optimistic planning. During the planning there had been tasks we predicted but we did not drill enough. One of the bases to estimate with the maximum precision possible is to divide bigger tasks in smaller ones because it is easier to estimate small and specific tasks than complex tasks which sometimes are also not well defined. We estimated too big tasks and we did not realize that these tasks imply "sub-tasks". When we have had to implement these wrong estimated tasks we have found that these tasks imply more work than we have predicted.

Related with the previous point, and as it is reflected in the Gantt diagram, is the fact that the estimation we did for the 30 days of GEP. We expected to be able to dedicate time to both, GEP and the project development. Unlucky, the workload of GEP forced us to do not dedicate time to the development and postponed until we finished GEP.

The last important problem has been the adaptation we have to do of the *scrum*. Currently we found with a functional API in a basic level but without a functional app. If we would have developed using strictly *scrum* and considering that did not exist a learning curve of *Android* we would not have the API almost finished but we would have had functionalities fully ended. For example: currently users could be created, authenticate and interact with the API creating products. However, there is not any part of the interface available to be tested by the user. We have half of the product but this part cannot be showed to the user. If we would have applied *scrum* we would have this half product finished but distributed in a very different way. Maybe we would not have the creation of products, but we would have an application where the users could authenticate themselves, create shopping lists and share them; without products but we would have something to show to the user.

This last statement seems cool and maybe somebody could ask why we do not have made the project following the process on the previous paragraph. This answer is that in fact a learning curve of *Android* does exist and consequently, make the process as the previous paragraph is almost impossible. If it has been already costly develop only a part of the project, make decisions related with this part, dominate the technology and consider business functionalities focused on this part of the API, make this process for all the functionalities twice with different technologies and best practices would have implied a lot of more hours.

Finally, evaluate that with all this deviations we have had to adapt in order to continue with the project. In the project there are three basic points: the scope, the time and the economy. In our case the economic aspect is secondary and we substitute it for quality and resources. We consider that having as goal the best practices and the production of a quality code is fundamental to add also to the equation this aspect. Referred to the resources point, this tries to reflect the economic aspect: this project does not have any capital investment, so deeply the only resource we have is ourselves.

Having to adapt ourselves to some of this four points and having a limited time due to the deadline, the quality of the goals and the resources (because we are the only developers of the project), we were forced to reduce the scope. It is important to say that we have never considered reducing quality, because it was fundamental that all what we did had a high quality grade. It was our first goal of the project.

8 Economy management

8.1 Initial budget

In order to perform the initial budget we were based on two premises.

The first one was not to exceed the limit of 15000€. We consider this figure is acceptable for a prototype. The second premise was based on the salaries. The wages of a developer are more or less 40,00 €/hour. However, the project would not be developed by a senior programmer. It would be managed by someone in a range between a junior developer profile because of his/her lower salary. The same considerations were extracted in the rest of roles we included in the project. All of them had a profile among a range between Fellow and Junior programmer. The other side of that role was the time necessary to perform a task: we deemed the time performed was longer. In short The cost per hour of the profile was cheaper whilst it took longer to carry out the task.

8.1.1 Cost identification

Table 3 shows the human resources that we deem necessary to carry out the project and the costs associated with each one.

Role	**Cost** €/h
Junior Analyst	25,00
Junior architect	30,00
Junior developer	20,00

Table 3: Human resources costs of the project

We could have added very specific roles such as Project manager, Usability expert or System specialist but we considered appropriate to differentiate only these three. During the solving problems stage we were able to see three different characters: .the one who analyzes the problem and search for possible casuistry, the Analyst; the one who designs a solution to the problem, the architect; and who develops it he will create the test and will track the integration with existing code. Although the three roles may be required for each project task, some are do not need essentially in some tasks. For example, one analyst was not necessary to create a user authentication. The issue is relatively simple. However, it was required to make the early decisions where it must be taken into consideration a wide range of situations and cases. So we decided to use the architect and the analyst in the tasks where their contribution was particularly needed.

The cost of human resources planned for the project tasks are shown in the table 4 and they are separated by periods identified by their corresponding colour.

Stage	**Cost**
Preparation and evaluation	3.505,00 €
Development API	3.680,00 €
Development App	2.790,00 €
Final delivery and documentation	1.480,00 €
	11.455,00 €

Table 4: Analysis of the initial human resources costs of the project divided by stages

Having seen this and referencing the partial table 4 and the complete information shown in the annex E we calculated that the human costs of the project would be: 11 455€. We refer to the section "scope" to justify the cost of the final stage of delivery and documentation.

Table 5 shows the costs related to software and licenses used during the project. Everything that is not specially mentioned is free software without any kind of restriction. Those who are usually paid but certain conditions have allowed to be free are: all *Microsoft* licenses that have been obtained thanks to the software distribution service of the UPC [18]] with a student account which allows you to access to the full functionality of the products, and the free license of the *PhpStorm* which is also a student account from any university by the period of one year

Software	**Cost**
Microsoft Project	0€
Microsoft Office Power Point	0€
ShareLatex online	0€
PhpStorm i *Android Studio*	0€
VMware	208,22€[19]
Git	0€
Laravel Framework	0€
Testing tools	0€

Table 5: Analysis of the software costs of the project

The estimated total cost of the software has been : 208,22€.

In table 6 you can see the estimated costs resulting from necessary project infrastructures.

In these facilities there are costs as the same computer, external services or basic infrastructures such as electricity or Internet connection. Virtual Machine infrastructure is necessary but, nevertheless, it does not have cost because it has already been contemplated in the *VMware.* license.

Infrastructure	**Cost**
Computer with *CPU* i5 and 8GB *RAM*	68,32€
Virtual machine Ubuntu 14.04 LTS	0€
Google developer account	25€
Git repository and *Bitbucket* account	0€
Internet connection	63,40€
Digital Ocean server	25€
Electricity	38,08€

Table 6: Analysis of the infrastructure costs of the project

We proceed to explain each of the associated costs. The total price of the computer used is 1500€. This price is not justified by the needs of the project: with a cheaper computer we could have worked but it was the equipment we had in the time of the project. The equipment cost is calculated by using the linear coefficient minimum repayment fixing by the Tax Agency State[20] which stating that this ratio may be 25%. Considering the price, the coefficient and that is devoted four hours a day over the course of approximately 133 days, we estimated the **hardware** amortization is: (1500 * 0.25 * 133 * 4) / (8 * 365) = 68.32 €.

The other cost calculation is simpler. For a *Google developer* license we had been pay 25€and it's a lifetime license. We can calculate which percentage of amortization corresponds to this project but considering it was a license that it has been buy again, we decided to load all of its cost in the project.

The third cost is also related to the server directly. From September to February and we estimate that we would need to have a pre-production server for testing, configurations, etc. In addition, we wanted to use this server later to connect our applications in beta to an environment outside of the development. The equipment costs 5 €every month [21], multiplied by 5 months it is €25.

Entering in the indirect costs we started by the Internet connection section. It was calculated from the cost of the contract line which is 29 €a month and considering all the assumptions made in the calculation of the computer depreciation we get a final cost of : (29 * 12 * 133 * 4) / (8 * 365) = 63.40 €.

Finally, the section of electricity (also belonging to indirect costs) was taking considering the cost of energy invoiced like a final consumers which has a price of 0.12696 €/kWh. The average power required by all computer components is about 450W and the sum of all peripherals reaches 600W, which multiplied by 500 hours of work, allows us to obtain a total consumption of 294kWh during the project. Multiplying the energy and the price mentioned above, we get the cost for the electricity section: 38, 08 €.

The total cost of the infrastructures was estimated at: 219,80€.

8.1.2 Cost estimates

Having identified the costs separately, we made an initial approach of the project cost with taxes included (Table 7).

Type of cost	**Cost**	**Total**
Human resources	11.455,00€	
Software	208,22€	
Infrastructure	219,80€	
		11.883,02€
IVA		2.405,54€
		14,378,46€

Table 7: Summary of total costs

First we can see that our budget is below of 15.000 €initials that we had set as a limit. After obtaining the final budget, we try to anticipate any deviations that may arise.

Before proposing any contingency plan or contingency item we want to show our disagreement with having a closed end with delivery targets. We are in favour of methodologies agile which propose objectives or time, but never both together, thereby facilitating customer understanding and can to adjust costs in every sprint keeping the client informed.

That said, and as there was no choice but to do it with a closed time and goals,, we decided to add to the end of each stage in the budget, a little time to review the features, a line item contingency. In the event that we finished the job on time we could improve the quality of our code, making refactor or progress in new functionalities. If there was an unexpected incident we would have reserved this time to solve them.

Apart from this line for unexpected items, during planning we detected critic points in the course of the project that should be taken into consideration. We believe that these critical tasks -market in red in annexes C and E- it could be a delay due to its complexity, so we decided to create a small contingency plan (Table 8). Obviously the first critical task –initial decisions- is not covered by this plan because the task had been completed by the time of planning.

Accepted risk	**% Frequency**	**Hours**	**Estimated cost**	**Risk exposure**
Implementation of the creation and modification	60%	12	240,00 €	144,00€
Implementation of simultaneous edition case	70%	20	400,00 €	280,00€
Creating the logic of adding products to the list	50%	12	240,00 €	120,00€

Table 8: Assumable risks and cost

To calculate the estimated cost of each risk we multiplied the time we thought that we need to solve the task as a price junior developer.

The total amount of this contingency plan rises in 544 €the line item of human resources and the final budget was estimated at:

Tipus de cost	**Despesa**	**Total**
Human resources	11.999,00€	
Software	208,22€	
Infrastructure	219,80€	
		12.427,02€
IVA		2.609,67€
		15.036,69€

Table 9: Summary of the total costs with contingency plan

9 Architecture and design considerations

9.1 Distributed app

As we have mentioned previously, we believe the solution to solve the problem could be store the information in the cloud. This means that if users want to interact with our cloud service through an app, our system will be distributed between client side and server side.

Considering this conclusion, we asked ourselves what responsibilities will have every part. The answer was that all the logic business and checks should be, at least, on server side. This is a public service that anyone could access. Therefore, the app checks and restrictions must be implemented on server side due to this requirement.

The other question that emerged was whether or not the application had to contain business logic or checks. We evaluated that the app could contains checks to minimise Internet connections, to save resources and improve the response time of the application getting better feedback from user. Concerning to add business logic to the application we thought that it was necessary (in some sites) but we decided to add the minimum possible, trying to separate our app from devices, because if some day we want to upload an app compatible with iOS devices we would have to maintain two logic business in different platforms.

This is the architecture's scheme;

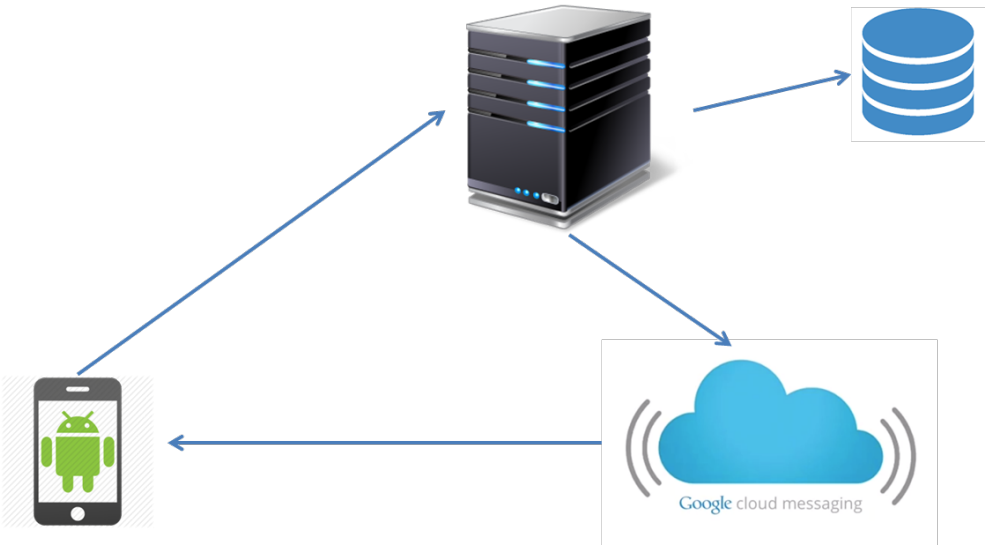

Figure 6: App architecture

In conclusion: we considered to develop a server side which follows the REST standard in most of the routes and create specific routes to carry out others complex business processes. Regarding the app, we thought it as a consumer client of the API, with the minimum knowledge of the business. It knows the routes to create, update and get the resources and the business routes to do more complex actions, like add a user to a list.

As we can see the server integrates itself in two more parts: a database, which at this moment is at the same server of the API, and the messaging service of *Google*. Due to we depend on two external components, we chose to apply *Ports & Adapters* philosophy. This, is going to be explained in the next section.

9.2 Hexagonal architecture

As we said our back-end integrates with other components that are not in our *Core Domain*. To solve problems related with integration with thirds software we have used the hexagonal architecture. This architecture, also known as *Ports & Adapters*, emphasizes the use of interfaces to communicate between layers, making easy establish layers between our software parts.

This is the architecture scheme

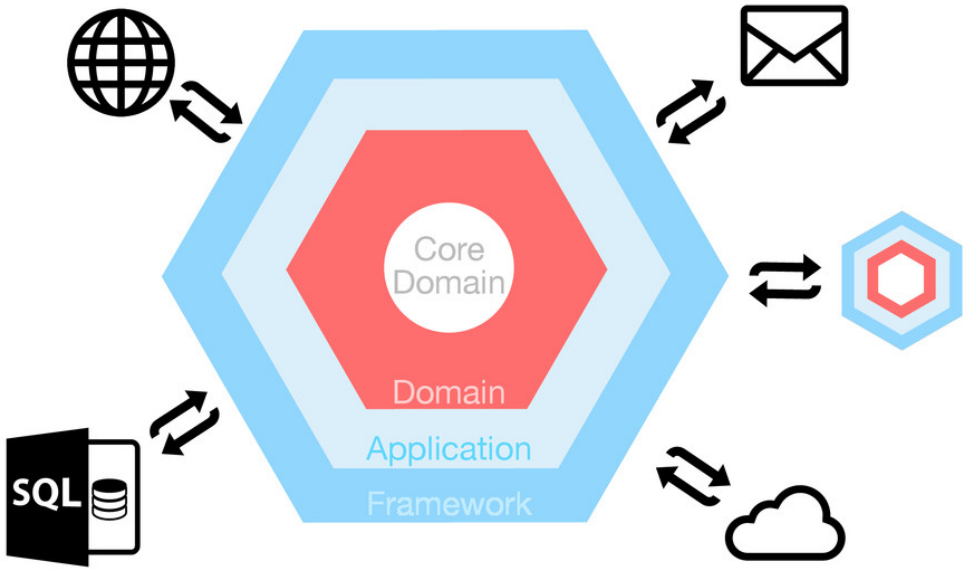

Figure 7: Hexagonal architecture

The first layer and outermost it is the framework layer. It is a layer which makes easier to develop and solve problems like the reception of HTTP requests, the sending of mails, the driver connection with database or the connection with other services. This layer is so useful that can become a headache over time. If we do not become independent from it, we could have important problems if some day decide to change the framework or libraries. What propose hexagonal architecture is to become independent from framework code using interfaces. If some day we want to change the framework, our business logic will not be affected for this change, and we only must develop new classes which implements the interfaces and swap our implementation for the old framework classes.

The application layer contains the services, the subscribers and the listeners, events and command bus. It is an intermediate layer that knows how to communicate the framework with our domain layer. Finally our domain layer where there are our business logic.

Moreover of proposing the use of interfaces to become independent of everything that was not our core or domain, it propose the *Command Bus* pattern. We thought about if we should use this pattern, because it looked interesting but we did not see how to apply it. Finally after some design and implementation failed proofs we decided not use it, and look it again after deliver the project. We preferred develop the project using an adaptation of this pattern. We decided to create a Service layer which will be the responsible to interact between our domain models. This would refactor a possible refactor in the future.

The main problem we found at *Command Bus* pattern was how to became independent of framework. We did not found the way to make it work being absolutely independent from the framework.

9.3 Rich models - anaemic models.

Previously we have mentioned this disjunctive and the visions about it. To sum up, there are opinions which consider anaemic models an anti-pattern which should not be used. When we started to develop we used this models but quickly we read about this issue. We did not find the way to fit rich models in our design and we did not understand the reason. As we were reading we found other opinions in favour of anaemic models, taking into account that there were situations that could require those model types. With that variety and the difficulty to puzzle the rich models in our implementation we decided to bet for anemic models.

Now, with perspective and after having read a little bit more about this issue we consider that, given the architecture on which we bet, it was not a bad decision. We see that probably we will have to rethink the fact of using rich models in case we want to apply hexagonal architecture

9.4 Device

At the moment we decided to create an app in order to interact with the server, we were fully conscious that this implies several limitations. The fact of working with a client with these characteristics forced us to take into account a series of facts.

The first and positive consideration was the fact we had the support of *Google Cloud Messaging* in order to notify the users the actions about the lists from other users. This made easier some of the functionalities.

The second and negative consideration was the data use. The fact of having a mobile device limited us both speed transfer and.usage data volume. We could.not think that the app would interact equally with the API as with a computer, we had to limit in the measure of possible the sending of the data and do it at the way as compact as possible. This problem supposed too much effort and time and finally we abandoned this issue.

The third consideration was the usage. Initially when we thought the non functional requirements of the app we realized we would need to think carefully which one we would want that was the flow which the user would have to interact with the app. In a screen with little dimensions the interaction is limited, therefore we should be careful with the interface design.

Finally, and an issue we decided to not include in the project was the fact of controlling when there was a connection and when not Initially we planned different ways where the app itself would manage the warning to the server if the connexion was lost and how the server would solve the delayed updates. The complexity of this problem brought us to decided not to include it in the project.

10 Back-end Development (API)

10.1 User Stories

As we said, we did not intend to exercise as a client, therefore we did not have the requirements established. When we saw we should think and act in accordance to this role, we decided to work with a nearer agile methodology, through user stories where we would define our functional requirements with natural language. Even though the power of this methodology canot be appreciated in this case and could be more visible in other cases where the functionalities are nearer to the final user, we decided to apply it to API.

All the routes (unless otherwise specified) will have the following format: *HOST/api/v1.0/*. In case of error, the answer's format will contain an error and information field.

Create user

Justification: it is necessary to create the new users when they want to register in to the system

Acceptance criteria:

- ☐ Anyone with the app will be able to create a new user on the system
- ☐ The route to create them will be a POST action to */users*.
- ☐ The input must contain two fields: *user*, *device*.
- ☐ The *user* field must contain: *name*, *surname*, *email*, *password* and *language*.
- ☐ The *device* field must contain: *device_token* and *platform*.
- ☐ If the input is correct, the server will return a 201 code with an object with these fields: *id*, *name*, *surname*, *email* and *language*.
- ☐ If the input's format is not correct the server will return a 422 code with an error message *Invalid request format*.
- ☐ If the input's format is correct but some fields are missing or the field's format is not correct, the server will return a 422 code and will show in a *information* field which are the wrong fields.
- ☐ If *email* already exists server will return a 409 code with an error message *User already exists*.

Login

Justification: to use the app, users must be logged. To identify themselves, they need a token and this action return it.

Acceptance criteria:

- ☐ Anyone with the app will be able to login.
- ☐ The route to login will be a POST action to */login*, without */api/v1.0* prefix.
- ☐ The input must contain to fields: *x-user_email* and *x-password*.
- ☐ If the input is correct the server will return a 200 code with a field *token* containing the token to do the requests to API.
- ☐ If the input is incorrect because the request is malformed server will return a 422 code with an error message *Invalid request format*.
- ☐ If the combination of *email* and *password* are not correct server will return a 401 code with an error message *Wrong username or password*.

Authentication layer

Justification: to use the app users must be logged.

Acceptance criteria:

- ☐ All the requests that need authentication must be redirected to this layer which will validate it.
- ☐ The request that need authentication must contain the *USER-TOKEN* header with the token gotten with a *login*.
- ☐ If the token is valid this layer will allow the request.
- ☐ If the token is not valid because not exists, server will return a 401 code without message.
- ☐ If the token is not valid because it expired the server will return a 403 code with *Token expired* message. A token expires when it is unused for 30 minutes

Update a user

Justification: should be allowed to update some users fields, because if users make a mistake on the registry, they will need to change the information

Acceptance criteria:

- ☐ Only the user can update his own information.
- ☐ The route to update it will be a PUT action to */users/id*.
- ☐ The fields *email* and *password* could not be updated through this route
- ☐ Then input must contain the fields: *name*, *surname* and *language*.
- ☐ If the input is correct, the server will return a 204 code without response message.
- ☐ If the input's format is correct but some fields are missing or the field's format is not correct, the server will return a 422 code and will show in a *information* field which are the wrong fields.
- ☐ If the input contains the fields *password* or *email*, server will return a 403 code without message.
- ☐ If some user tries to update another user's information, the server will return a 404 code without message.

Get a user

Justification: we need to get the user's information to display it in the app

Acceptance criteria:

- ☐ Only the user can access to his own information.
- ☐ The route to get it will be a GET action to */users/id*.
- ☐ If the input is correct, the server will return a 200 code with an object with these fields: *id*, *name*, *surname*, *email* and *language*.
- ☐ If some user tries to get another user's information, the server will return a 404 code without message.

Create a shopping list

Justification: it is the base of our system. Currently it only allows the creation of empty lists because this is how we had originally planned the app's flow

Acceptance criteria:

- ☐ Any logged user will be able to create a shopping list.
- ☐ When a user creates a list, he becomes its administrator.
- ☐ The route to create it will be a POST action to */lists*.
- ☐ The input must contain two fields: *name* and *products*. At present, *products* will be considered always empty.
- ☐ If the input is correct, the server will return a 201 code with an object with these fields: *id* and *name*, *admin* which contains an object with the administrator of the list, *products* which contains an empty list of products and *users* with the same content as admin.
- ☐ If the input's format is not correct, the server will return a 422 code with an error message *Invalid request format*.
- ☐ If the input's format is correct but some fields are missing or the field's format is not correct, the server will return a 422 code and will show in a *information* field which are the wrong fields.

Add an user in a list

Justification: it is one of our distinguishing features respect of other systems in market. Lists can be shared and users can use the email as identifier of other users.

Acceptance criteria:

- ☐ Users will be added using the email. Only registered users can be added.
- ☐ Only the administrator user will be able to add a new user in a list which he is administrator
- ☐ The route to create it will be a POST action to */lists/id/add-user*.
- ☐ The input must contain a field *user_email*.
- ☐ If the input is correct, the server will return a 200 code and the server will return the same response as *Creació d'una llista de la compra*. In this case will return the products (if it exists) and the added user, will be in users who belongs to list in *users* field.
- ☐ If the email does not exists, the server will return a 409 code with an error message *User not found*.
- ☐ If an user tries to add another user in a list which is not administrator, the server will return a 404 code without message.

Get the lists of an user

Justification: we want to show all the user's lists in the app's home screen. We need this endpoint for a business requirement.

Acceptance criteria:

- ☐ Only the user can access to his own lists.
- ☐ The route to get it will be a GET action to */users/id/lists*.
- ☐ If the sent identifier exists, the server will return a 200 code and the response will contain an array with the lists. The lists will have the same content than *Creació d'una llista de la compra*. In this case will return the products (if it exists), with these fields for each product: *id*, *name*, *description*, *image*, *quantity* and *isBought*.
- ☐ If some user tries to get another user's lists information, the server will return a 404 code without message.

Get the app's products preloaded

Justification: we want to offer our users a set of preloaded products in the app in order to make it easy add new products, saving time by introducing data

Acceptance criteria:

- ☐ Any logged user should be able to get app's products preloaded.
- ☐ The route to get it will be a GET action to */app-products*.
- ☐ The server will return a 200 code with an array containing the products that previously we introduced in our database. Products will have these fields: *id*, *name*, *description* and *image* which will be an identifier of the picture.

Add a product to a list

Justification: basic feature to make sense to the app

Acceptance criteria:

- ☐ Any logged user who belongs to the list can add a product.
- ☐ The route to create it will be a POST action to *lists/id_list/products*.
- ☐ The input must contain these fields: *name*, *description*, *image*, *quantity* and *isBought*.
- ☐ The server will return a 201 code and the response will contain all the fields mentioned in the previous point plus *id*
- ☐ If the input's format is not correct, the server will return a 422 code with an error message *Invalid request format*.
- ☐ If the input's format is correct but some fields are missing or the field's format is not correct, the server will return a 422 code and will show in a *information* field which are the wrong fields.
- ☐ If an user tries to add a product to a list which not belong, the server will return a 404 code without message.

Update a product

Justification: core feature to allow our users mark products as purchased or change some attributes like quantity or description.

Acceptance criteria:

- ☐ Any logged user who belongs to the list can update a product of it.
- ☐ The route to create it will be a PUT action to *lists/id_list/products*.
- ☐ The input must contain these fields: *name*, *description*, *image*, *quantity* and *isBought*.
- ☐ If the input is correct, the server will return a 204 code without response message.
- ☐ If the input's format is correct but some fields are missing or the field's format is not correct, the server will return a 422 code and will show in a *information* field which are the wrong fields.
- ☐ If an user tries to update a product to a list which not belong, the server will return a 404 code without message.
- ☐ If an user tries to update a non existent product, server will return a 404 code without message.

Remove a product

Justification: in addition to mark the products as purchased, we offer to delete the product from list

Acceptance criteria:

- ☐ Any logged user who belongs to the list can delete a product of it.
- ☐ The route to create it will be a DELETE action to *lists/id_list/products/id_product*.
- ☐ If *id_product* exists, the server will return a 204 code without response message.
- ☐ If an user tries to delete a non existent product, server will return a 204 code without message.
- ☐ If an user tries to delete a product to a list which not belong, the server will return a 204 code without message but will not remove the product.

10.2 Non functional requirements

Usability

☐ The system will support three languages: catalan, spanish and english.

Availability

☐ The system will be always available as long as platform where it is deployed is available.

Security

☐ Before doing a deploy on production, the tests will be executed.

☐ The system should be hosted in a server with the following security measures:

- *Root* account disabled.
- Access using user and password via SSH filtered by IP. Only can be accessed from some computers.
- It will have protection against brute-force attacks via SSH (*fail2ban* for example).
- Periodic security's upgrades.
- The database access will be blocked for all non local ports.

☐ The system will not show any information of itself. For example it will not reveal which identifiers exists and which not.

☐ An user will not be able to access information that business rule not allow him.

Robustness

☐ The system will not showa stack-trace if there is an error.

☐ All the errors will be treated internally and it will be given an answer (previously thought) to the request.

Performance

- ☐ The response time may not exceed 500 ms.
- ☐ The time of unit tests can not exceed 100 ms for test.

Capacity

- ☐ The system will initially support 1,000 users.
- ☐ The system will initially support 10,000 purchase products.

Maintenance

- ☐ Every feature must have his own unit and integration tests.
- ☐ All the new features must follow the previous requirement.
- ☐ The system must be compatible with two previous versions.

Integrity

- ☐ Once a week there will be a backup.

Quality

- ☐ All the features will be free of bugs and will can be proved with automated tests.
- ☐ All the features must be implemented using TDD.
- ☐ The code will have 85% or more test coverage.
- ☐ The integration with services will be done through interfaces and the coupling will be the low as possible.
- ☐ SOLID principles will be respected.

10.3 Conceptual model

Given the requirements, the conceptual model we have worked is the following.

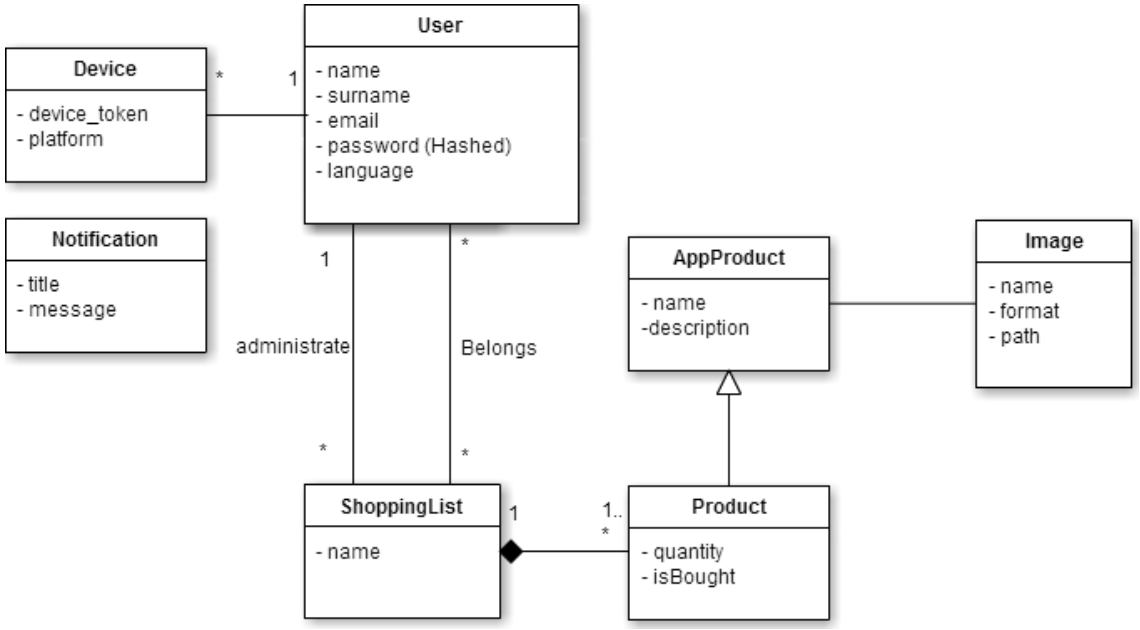

Figure 8: Conceptual model

Although notifications is not an entity currently related with other system entities there is in the scheme for two reasons. The first one is that we needed a data structure to be used by messaging services, because we did not want use *strings* without structure. We preferred to have an entity accord with our domain feature. The second is that in a future we expected that notifications may be an entity related with users, in order to make it easy and relying on the first reason we decided to add it in our domain's entities.

We also remark that we wanted include *Category* entity which would be related with *AppProduct*. However adding this entity implies a high complexity especially focused on the app. Add it only on server was not hard but it did not have sense. Thus, to develop something that will not be used on the app in the first version, we preferred focus on other tasks.

10.4 Database's model

The figure 9 shows the database's scheme. Arrows indicate the flow of the foreign key: *users_tokens* table has a foreign key which reference *id* of the table *users*.

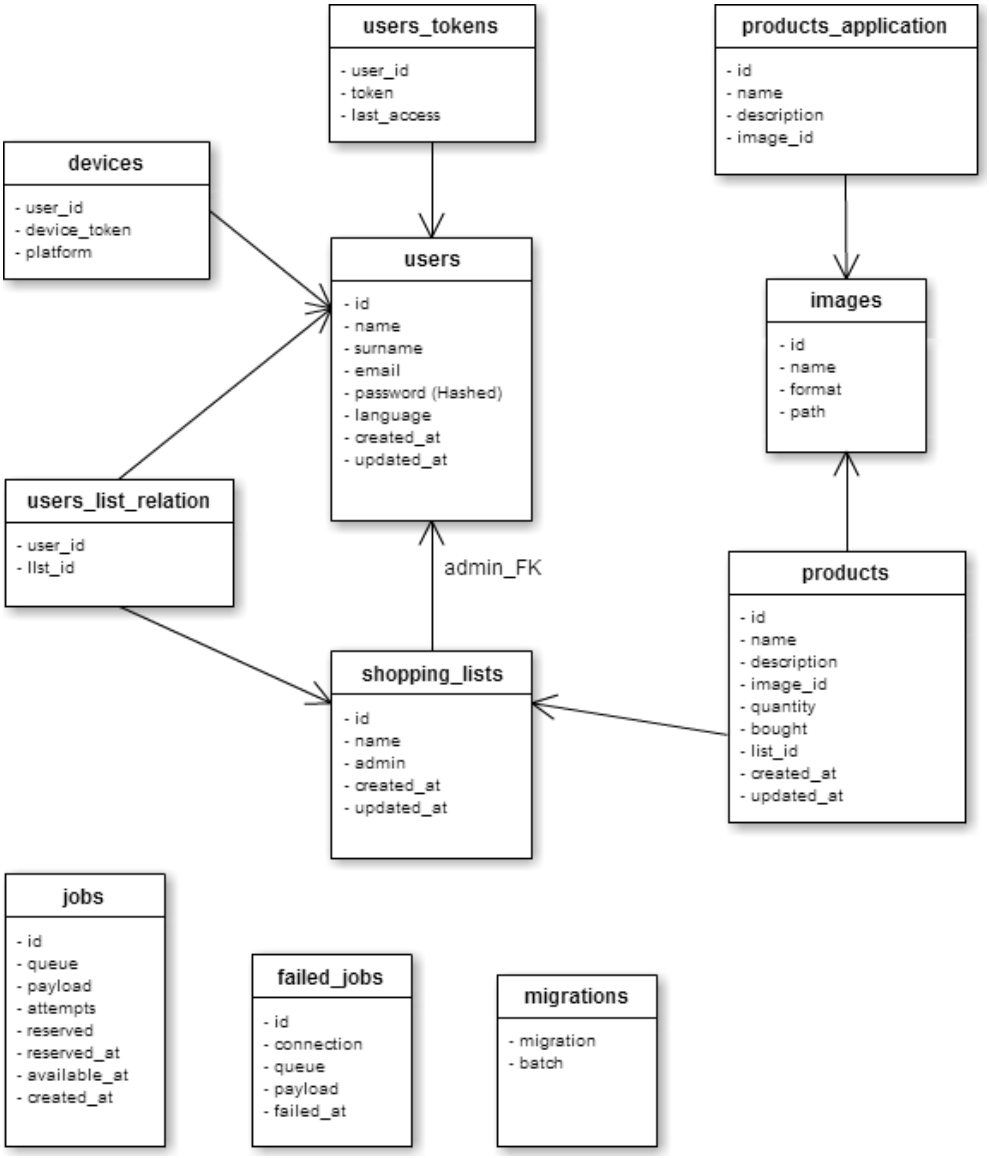

Figure 9: Conceptual model involving database's relations

In this scheme we can see three concepts that we have not introduced yet, and we want to do it in this section.

The *migrations* table is a table which enables us to control the database state. In order to advance the project and deploy it easy in any environment, we decided to use the framework's migration tool. This allows us to make changes or rollback it in an easy way in case we deploy something with a bug on production.

The other two tables exist due to queues system. During the design of the notifications system, we thought on this case: a big volume of notifications to send in our system, which was blocking designed initially, would induce a block the HTTP requests and provoke a timeout error. For this reason we implemented a queue system to add asynchronous tasks like is sending a notification.

11 App

As we said, this part of the project has not been developed due to lack of time. We did an optimistic forecast and this has been the result. Nevertheless, we worked in this part, less than the server part, but in section we will show you the results that we got.

11.1 Design and navigation

During the development of the project we saw that we would not have a client who made decisions, we designed ourselves the mock-ups. This was an essential work because we needed to know what app needed store on database and what needed to get from server. We have not specially worked on usability, just have we tried to do it as functional as possible and make it easy to use.

Figure 10: Register screen

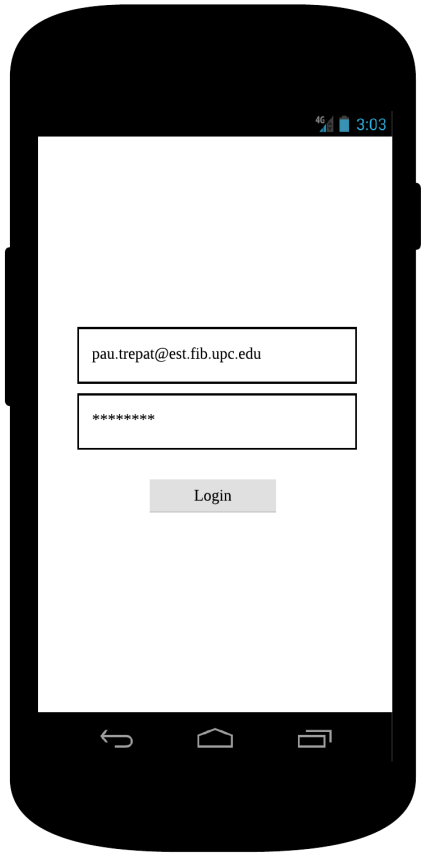

Figure 11: Login screen

Once the first register or login is done, these screens are not be shown anymore.

The screens' design is plain. The button with symbol [+] is used to add new lists in the screen shown in the figure 12 and to add products in the figure 13. We think the other features follow the most common standards in mobile apps (trash and a list of items) and do not require further explanations.

Regarding list of products (figure 13) the box which there is on the right would show if the product have been purchased or not. Nowadays we have not found find a better option to show to user this information, but we consider this is an usability problem, and we have preferred work on it in the future.

Finally regarding the screen described in the figure 13 includes the button to share the list with other users. Although we have not thought about how will be the dialogue format that is going to be shown we know that will use the user's email to identify him. A simple text box will be enough.

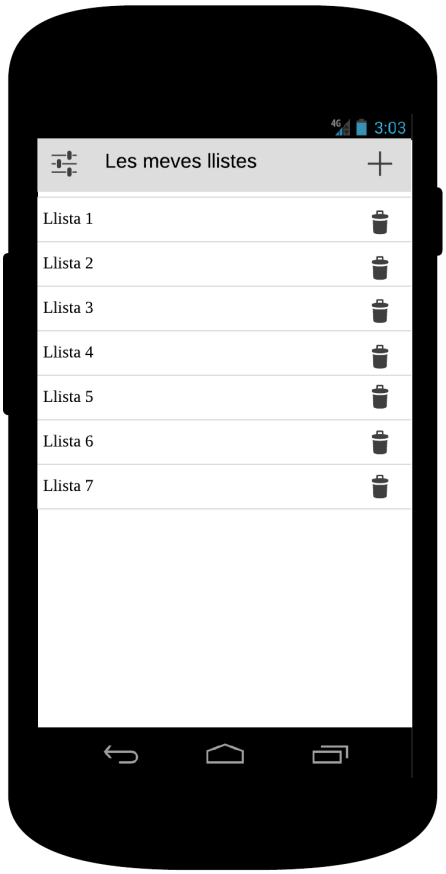

Figure 12: Home screen

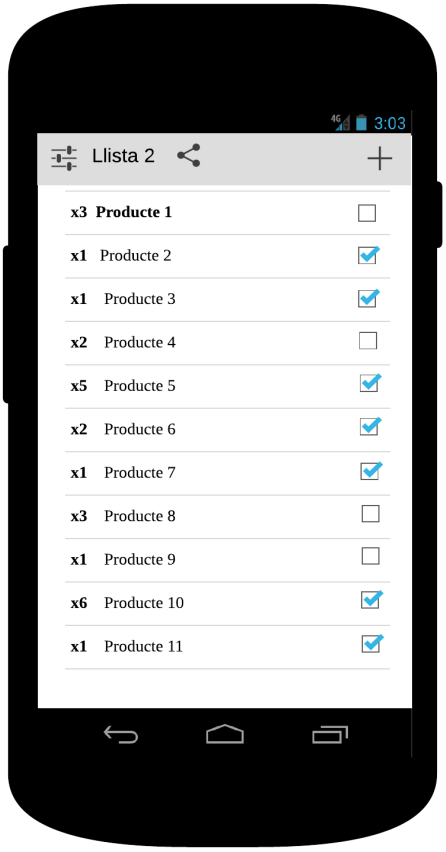

Figure 13: Shopping list's products

Figure 14: Product's information screen

To usability issues we decided to include the product's delete button on this screen. We believe it is a feature that will not be used often and where we have placed it will help the user without make it hard the use of interface.

11.2 Patterns

Proxy

Although we did not develop a solution, we have thought about how to decouple our domain layer of how data is persisted. It is not trivial because we have seen reading about *Android* and asking to *Android* developers with some experience that *Android* has a high coupling. But we think using a proxy pattern we will solve the problem.

The proxy pattern is a software design pattern which its purpose is to be the intermediate between objects' communication. There are different reasons like security but, in our case, we need proxy pattern to solve the problem that we mentioned where are the data stored.

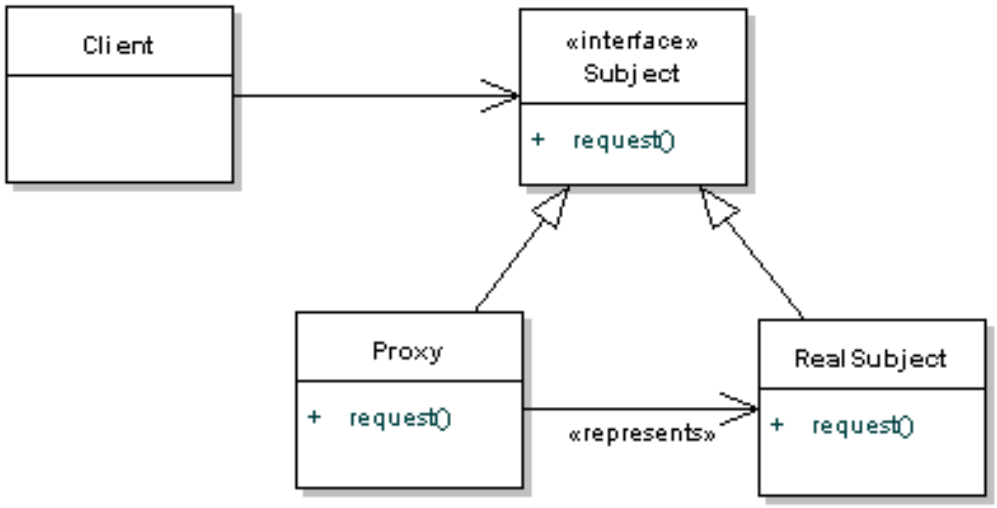

Figure 15: Proxy pattern

Initially we had doubts about how to use this pattern and how to build our domain, but after read about anemic and rich models, we understood that this was a case where we could apply a rich domain design. The own objects would implement domain's functions like adding a product to a list, and through the proxy pattern, we would apply changes on the server.

```
interface IShoppingList {
    public void addProduct (Product p);
}

class ShoppingList: IShoppingList {
    private ArrayList<Product} list;

    public void addProduct (Product p) {
        list.add(p);
    }
}
```

This code could be of any software without any persistence, just a domain function. We could choose not develop the interface and just have the *ShoppingList* class.

```
class ProxyShoppingList: IShoppingList {
    ShoppingList sl;

    public ProxyShoppingList(IShoppingList sl) {
        this.sl = sl;
    }

    public void addProduct (Product p) {
        sl.addProduct(p);
        updateOnServer();
    }

    private updateOnServer() {
        //PUT sl ...
    }
}
```

In the code above, we can see where the proxy pattern makes sense. This implementation shows how we can reach a low coupling between the domain and the persistence layers independently of platforms or architecture (distributed - centralised). It is only necessary that the proxy receives an object which implements *IShoppingList* interface.

12 Sustainability

To calculate the sustainability impact we will use 3 dimensions: environmental, economic and social. Each one of this dimensions is counted in the table 10 on a score of 1 to 10 (1 minimum, 10 maximum)

Sustainable	Environmental	Economic	Social	Total
Planning	Resource analysis	Economic feasibility	Improved life's quality	
Rating	6	5	7	18
Results	Resource use	Final cost vs estimate	Impact on environment	
Rating	8	-5	10	13
Risks	Environmental damage	Adapting to change	Social damage	
Rating	0	0	0	0
Total	31			

Table 10: Rating sustainability project

12.1 Environmental

This project haven't had large environmental costs. The major resources have been a computer and an Internet connection. In the original cost estimate, we took into consideration the server hosted in *Digital Ocean* but there has been no impact, because it has never turned on.

Regarding the issue of computer's hardware, we acknowledge did not use the most efficient but it was what we had previously bought and it was not purchased to develop this final degree project considering the environmental impact.

The only part that is particularly positive and directly related to the environment is the amortising food resources. Prevents two people buy the same food having more than necessary and if these products have an expiration date or cannot stay fresh for too long, avoid throwing them.

12.2 Economic

We consider the economic part is the part with less impact. Because it was a prototype and we cannot think about an useful life for this product neither benefits,we thought that these are an important reasons to decrease its importance. Although, we need to consider if we keep developing this project in a future we will have a major part of the project (back-end) and it will be less expensive.

One of the reasons that made decrease the economic impact was the learning curve of technologies like frameworks, design and think about possible issues of the architecture and work following the best practises for each task. This made slow down our development bringing less features which meant a lower software value, although we reduced significantly our technical debt.

Concerning the estimated costs and real costs, we did not estimate it well. The estimated costs included back-end and app development but with this resources we only could develop the server side. As we have said this slow development provides us a cohesive and robust code which allows reduce the technical debt and the impact of future changes. We assigned a negative value because we rate from perspective of a hypothetical client.

After being at an unexpected situation such as not having a product owner, we decide become our product owner taking the role and go ahead with the project. That is why we think that our adaptation to changes has been quite good.

12.3 Social

Finally we think that social dimension is the best rated. Although its impact on the number of initial users may be low because it is a prototype, will allow us fit the requirements of potential end users, getting a better reputation.

We believe this system will help the users, allowing them improve their daily and weekly purchasing processes saving time and resources in this basic activity.

Another social advantage that the project brings are the knowledge acquired by the author. All the time used to learn and consolidate the concepts like best practises, TDD and design patterns has resulted in improvement of author's value on workplace because is a well-trained professional with an added value. The knowledge of concepts like technical debt or code smells and how to fix them implies a better assessment from enterprises in relation to other recent graduated students.

13 Conclusions

The main idea we have learned after months working is that a good software development needs time and resources. In perspective we think that we have finished and delivered more complex practices in degree but with a high technical debt, hard to maintain and apply new features. To find a solution to a relatively complex problem, implement it in any language and execute it, a second year student is able to do it. However, develop a system thinking about to maintain over time, all the consequences of each decision, implement the solution, designing and implementing tests and finally make a good deploy flow requires time.

Rene Lavand, a magician, said that: *"El conocimiento tan sólo es un rumor hasta que llega al músculo"*. It says that the knowledge it is just a whisper until it reaches the muscle. We think that is absolutely true. Since we start develop we were convinced about what was correct and what wrong, and if we did not know, we searched it. But when we started write code, everything we knew was theory. Ideas, good intentions and large musings but we did not know how to apply it in our project. Over time, developing, and implementing all these ideas and best practises we have acquired fluency to analyse and think solutions.

The same happens with tests. At the beginning, develop following TDD philosophy has taken time. One of the difficulties has been to learn how to apply it. The most common question we had was: "What i have to test if i don't know what i want that it returns?". With time and practice we have understood how it works, how apply it and what gives us. Actually would be hard develop without tests,

One of the most important concepts that we have learnt is the technical debt. A nonexistent concept in degree, because the practises that we deliver end up, at best case scenario, in a code repository and because it is only evaluated if it works, best practises or design are not evaluated. The cost associated to bad decisions will always exists in a project with a finite time to end, we know. But we must minimise this costs, because it has the snowball effect and all the accidental costs that we add to the project could cause an unattainable cost. We know some apps that have been done again for it is unattainable maintenance cost:without test, without well defined domain entities and with a high coupling with thirds' technologies and libraries.

Analysing the development of the project we think that the bet in favour of user stories to define the API have not been as effective like would have been to define app's requirements. Because it is a more technical part and less close to the language of the user, it has missed the potential offered by this technique and user stories have been unnatural. We still defending that write excessive documentation is a waste of time and resources because the good code explains itself and it is more easy to understand if it is helped by tests. One of the reasons is tests need to change with code, or it will not be successfully executed. The written documentation not necessarily changes at the same time than code, and when two or three changes are applied, the documentation do not describe the code's behaviour .

But it does not mean that we do not believe in written documentation. We know that some projects with big teams or with the need to outsource some parts the documentation is essential. A developer who could talk with stakeholder, know what the customer wants. Nevertheless, if this possibility not exists, the developer needs some support for orientation and in this case we agree with the use of exhaustive documentation

Last but not least, to assume that this was an academic exercise. Our intention to have as goals the best practises and to design carefully every aspect of our system is noble, but the workplace the productivity has most value. In this project we have valued more the way than the result. Our intention was not finish the project to any cost and do a botch job. We wanted to build a reference for a future projects. We did not want open the project after a year and do not know how to continue the project like with other projects happens.

After carrying out the project, we have assimilated a set of knowledge and skills that next time we have to apply it, the time needed will be less. We could say this project have been like a big *kata* [22]. We have confirmed that the best way to learn is trying; trying, failing and fixing. And this project have been an excuse to do two projects that we wanted to do long time ago: make it easy the management of the shared shopping lists and start a project that allows us learn about real life projects.

We encourage everyone who reads this project to consider whether the methodologies we used are useful and if it is better to do small (or large) investment at a time to get better results in the future. The tests, whatever they are, will allow you to add new features and do refactor with security and without fear of breaking anything. TDD will help you to detect easily design error and will help to think about which are your features and his behaviour. SOLD principles are indispensable to write code with a decent quality and will reduce your technical debt. And ultimately hexagonal architecture will also reduce technical debt and will provide have low coupling with most of the dependencies.

14 Future work

Obviously, the first job is continue developing the application. Currently there are a solid base regarding back-end but UI has to be built. But is not the only task that we have to do. With this TFG we only secured bases to start the project, functional and methodological.

Although the design of the application architecture is designed, lack implement and evaluate it. In addition, continuing the working line, we have to expand our technical and design knowledge in the area of *Android*. Missing long way to go in this area. During the course of the project we feel our lack of knowledge in best practises, architecture and design in *Android* area.

Following with the application must address the issue of graphic design. If we want to deploy the application to the mobile stores, although as beta version, we should take care of the user interface, task that on this project we have not included.

Besides from tasks of the application we would have to deploy a small server to install Jenkins or Travis and work with continuous integration. Nowadays the project has been only developed by one student without production deploy. Automate execution of the tests and deployments would prevent human errors and would facilitate the work when the application is in production. Continuous integration is a world apart and is so extensive that it could be the center of a TFG.

We believe that we have to place on record the assessment we made of written documentation formally of the API, but as we said in our conclusions, we think that it is a waste of resources and because API was not public, we thought that was not necessary. Moreover, if some day API was to become public we should change the codes that are returned in some cases to make it more understandable and easier to use.

Finally, we applied most of the hexagonal architecture principles except *Command* pattern.We have been talking with different developer who have applied hexagonal architecture in their projects and has occurred the same situation: they applied most of the principles except this. Some due to complexity, others because the time effort was too great and others because they thought that would not help on developing. We think that we can improve as developers if we internalise and apply this pattern usually. So through this project we will try to work *Command* pattern. But because we have worked following TDD, we can do safety refactor because tests allows us to detect if we break something.

15 References

[1] Martin Fowler. *TechnicalDebt*. http://martinfowler.com/bliki/TechnicalDebt.html. [En línia; accedit 02-07-2015]. October 2003.

[2] Enrique Comba Riepenhausen. *Your application is not your framework*. https://speakerdeck.com/ecomba/your-application-is-not-your-framework. [En línia; accedit 10-12-2015]. December 2015.

[3] Wikipedia. *Naming convention (programming)*. https://en.wikipedia.org/wiki/Naming_convention_(programming). [En línia; accedit 25-08-2015].

[4] Samuel Oloruntoba. *S.O.L.I.D: The First 5 Principles of Object Oriented Design*. https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design. [En línia; accedit 01-09-2015]. March 2015.

[5] Scott W. Ambler. *Introduction to Test Driven Development (TDD)*. http://www.agiledata.org/essays/tdd.html. [En línia; accedit 25-07-2015].

[6] Chris Fidao. *Hexagonal Architecture*. http://fideloper.com/hexagonal-architecture/. [En línia; accedit 20-09-2015]. 2014.

[7] Alister Scott. *Introducing the software testing ice-cream cone (anti-pattern)*. http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/. [En línia; accedit 17-08-2015]. January 2015.

[8] Martin Fowler. *Mocks Aren't Stubs*. http://martinfowler.com/articles/mocksArentStubs.html. [En línia; accedit 07-08-2015]. January 2007.

[9] William Pietr Alexander Chaffee. *Unit testing with mock objects*. http://www.ibm.com/developerworks/java/library/j-mocktest/index.html. [En línia; accedit 30-09-2015]. November 2002.

[10] Martin Fowler. *AnemicDomainModel*. http://www.martinfowler.com/bliki/AnemicDomainModel.html. [En línia; accedit 24-10-2015]. November 2003.

[11] *Rich vs Anemic Domain Model*. http://stackoverflow.com/questions/23314330/rich-vs-anemic-domain-model. [En línia; accedit 25-10-2015]. April 2014.

[12] *The Anaemic Domain Model is no Anti-Pattern, it's a SOLID design*. https://blog.inf.ed.ac.uk/sapm/2014/02/04/the-anaemic-domain-model-is-no-anti-pattern-its-a-solid-design/. [En línia; accedit 25-10-2015]. February 2014.

[13] Sebastian Bergmann. *PHPUnit*. https://phpunit.de/. [En línia; accedit 25-06-2015].

[14] *Mockery*. https://github.com/padraic/mockery. [En línia; accedit 25-06-2015].

[15] Joshua Thijssen. *Idempotency*. http://restcookbook.com/HTTP\%20Methods/idempotency/. [En línia; accedit 11-11-2015].

[16] Roy Thomas Fielding. *Representational State Transfer (REST)*. https://www.ics.uci.edu/fielding/pubs/dissertation/rest_arch_style.htm. [En línia; accedit 05-11-2015].

[17] *OAuth Site*. http://oauth.net/. [En línia; accedit 05-11-2015].

[18] Universitat Politècnica de Catalunya. *Distribució de software*. https://upcnet.upc.edu/serveis/sistemes-dinformacio-de-la-universitat/tic/distribucio-de-software. [En línia; accedit 02-09-2015].

[19] Inc VMware. *VMware Workstation 12 Pro*. http://store.vmware.com/store/vmwde/es_ES/DisplayProductDetailsPage/ThemeID.29219600/productID.323427300. [En línia; accedit 10-09-2015].

[20] Agencia Tributaria. *MANUAL PRÁCTICO Sociedades 2014*. http://www.agenciatributaria.es/static_files/AEAT/DIT/Contenidos_Publicos/CAT/AYUWEB/Biblioteca_Virtual/Manuales_practicos/Sociedades/Manual_Sociedades_2014.pdf. [En línia; accedit 01-10-2015].

[21] DigitalOcean ™ Inc. *Simple Pricing*. https://www.digitalocean.com/pricing/. [En línia; accedit 07-07-2015].

[22] Dave Thomas. *CodeKata*. http://codekata.com/. [En línia; accedit 12-01-2016].

Glossary

A

Anaemic model

Software pattern used in domain design where domain objects contain little or no business logic (validations, actions..)

API

Application Programming Interface: Interface which specifies how to interact with a particular component.

K

Kata

Performed exercises to learn and practice, simple problems which the solution is known, but the important is not the resolution but apply specific knowledge and improve skills as developers.

R

REST

Representational State Transfer: is a style of architecture that follows a set of constraints oriented hypermedia systems.

Rich model

Software pattern used in domain design where domain objects contain all the business logic.

T

Technical debt

That concept refers to the consequences of an inaccurate design and over time, if it is not solved, increases.

A Initial planning

Nom	Inici	Fi
Fase 1 - Preparació i avaluació	**14/06/2015**	**14/07/2015**
Creació d'una màquina virtual i instal·lació del SO	14/06/2015	16/06/2015
Documentació tria i desplegament del framework	16/06/2015	22/06/2015
Instal·lació de dependències	22/06/2015	23/06/2015
Instal·lació de les eines de desenvolupament del servidor	23/06/2015	25/06/2015
Creació del repositori	25/06/2015	26/06/2015
Lectura de documentació relacionada amb seguretat	26/06/2015	28/06/2015
Creació d'una *wiki*	28/06/2015	01/07/2015
Presa de decisions inicials	01/07/2015	14/07/2015
Fase 2 - Desenvolupament API	**01/09/2015**	**27/10/2015**
Configuració de l'entorn de testing	01/09/2015	01/09/2015
Creació d'usuari [API]	02/09/2015	09/09/2015
Actualització d'usuari [API]	10/09/2015	14/09/2015
Elecció del mètode d'autenticació	17/09/2015	21/09/2015
Implementació del mètode d'autenticació	22/09/2015	29/09/2015
Preparació per a dispositius	30/09/2015	05/10/2015
Enviament dels emails per proposar col·laboració	28/09/2015	28/09/2015
Creació del compte de *Google Developer*	28/09/2015	28/09/2015
Disseny de la part de les llistes compartides	06/10/2015	08/10/2015
Implementació de la creació i modificació	09/10/2015	16/10/2015
Implementació del cas de l'edició simultània	19/10/2015	22/10/2015
Revisió i testing de l'API	23/10/2015	26/10/2015
Desplegament a pre-producció	27/10/2015	27/10/2015
Fase 3 - Desenvolupament Aplicació	**02/11/2015**	**16/12/2015**
Configuració de l'entorn (*drivers* i *JUnit*)	02/11/2015	04/11/2015
Instal·lació de les eines de desenvolupament de l'aplicació	05/11/2015	05/11/2015
Creació de pantalla de login	06/11/2015	10/11/2015
Creació de la pantalla principal	11/11/2015	12/11/2015
Proves d'integració amb el servidor	13/11/2015	13/11/2015
Creació de la lògica d'addició de productes a les llistes	16/11/2015	25/11/2015
Creació de la lògica de modificació de llistes	25/11/2015	11/12/2015
Revisió de les funcionalitats i correcte integració total	14/12/2015	16/12/2015

Nom	Inici	Fi
Fase 4 - Entrega Final i documentació	**17/12/2015**	**11/01/2016**
Prova pilot i resolució de problemes	17/12/2015	22/12/2015
Temps de marge per possibles desviacions	23/12/2015	25/12/2015
Redacció de documentació	28/12/2015	04/01/2016
Disseny de la presentació	05/01/2016	11/01/2016

Table 11: Planning project tasks

B Development tasks

Nom	Inici	Fi
Fase 1 - Preparació i avaluació	**14/06/2015**	**14/07/2015**
Creació d'una màquina virtual i instal·lació del SO	14/06/15	16/06/15
Documentació tria i desplegament del framework	16/06/15	22/06/15
Instal·lació de dependències	22/06/15	23/06/15
Instal·lació de les eines de desenvolupament del servidor	23/06/15	25/06/15
Creació del repositori	25/06/15	26/06/15
Lectura de documentació relacionada amb seguretat	26/06/15	28/06/15
Creació d'una wiki	28/06/15	01/07/15
Presa de decisions incials	01/07/15	14/07/15
Fase 2 - Desenvolupament API	**01/09/2015**	**06/01/2015**
Configuració de l'entorn de testing	01/09/15	02/09/15
Actualització de la wiki	02/09/15	02/09/15
Disseny d'autenticació	03/09/15	04/09/15
Avaluació del patró Command Bus	04/09/15	07/09/15
Avaluació models anèmics / rics	07/09/15	10/09/15
Segureta saturació de la BD	10/09/15	11/09/15
Creació d'usuari [API]	11/09/15	16/09/15
Valoració idempotency del update	15/10/15	15/10/15
PUT VS PATCH	15/10/15	16/10/15
HTTP resonses	17/10/15	17/10/15
Refactoring	18/10/15	19/10/15
Actualització d'usuari [API]	19/10/15	23/10/15
Lectura REST autenticació	23/10/15	24/10/15
Disseny de l'autenticació	24/10/15	25/10/15
Elecció del mètode d'autenificació	25/10/15	27/10/15
Implementació del mètode d'autentificació	27/10/15	30/10/15
Refactor test antics	31/10/15	02/11/15
Lectura documentació GCM	02/11/15	03/11/15
Preparació per a dispositius	03/11/15	07/11/15
Enviament dels emails per proposar colaboració	01/10/15	01/10/15
Creació del compte de Google Developer	09/10/15	09/10/15
Tests d'integració amb GCM	07/11/15	08/11/15
Disseny del concepte de llista	09/11/15	10/11/15
Interaccions de les llistes	11/11/15	12/11/15
Disseny BD de les llistes	12/11/15	12/11/15
Orientació a dispositus mòbils llistes	13/11/15	15/11/15

Nom	Inici	Fi
Disseny de la part de les llistes compartides	15/11/15	15/11/15
Obtenció de les llistes	16/11/15	18/11/15
Refactoring collections	18/11/15	19/11/15
Dubtes i proves Repositories	23/11/15	27/11/15
Implementació de la creació	27/11/15	02/12/15
Disseny del concepte de producte	02/12/15	03/12/15
Consistència entre productes app i API	03/12/15	07/12/15
Creació de productes de la aplicació	18/12/15	19/12/15
Creació de productes	22/12/15	23/12/15
Refactor per retornar llistes amb productes	23/12/15	24/12/15
Modificació de productes	02/01/16	05/01/16
Eliminació de productes	06/01/16	06/01/16
Fase 3 - Desenvolupament Aplicació	**14/12/2015**	**24/12/2015**
Lectura best-practices Android	09/12/15	11/12/15
Desenvolupament recepció push	14/12/15	17/12/15
Fase 4 - Entrega Final i documentació	**09/01/2015**	**18/01/2016**
Redacció de la documentació	09/01/16	18/01/16

Table 12: Actual development of project's tasks

C Gantt chart of initial planning

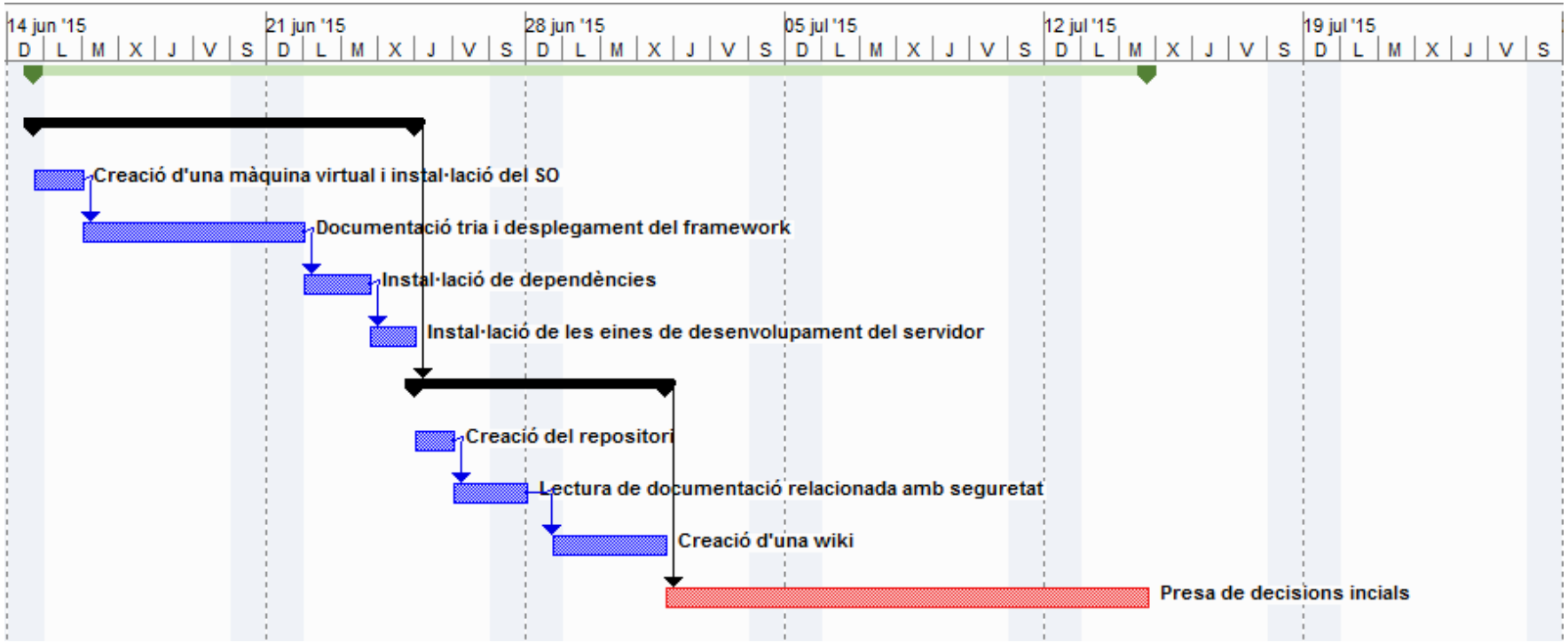

Figure 16: Gantt chart of the stage Preparation and evaluation of the project

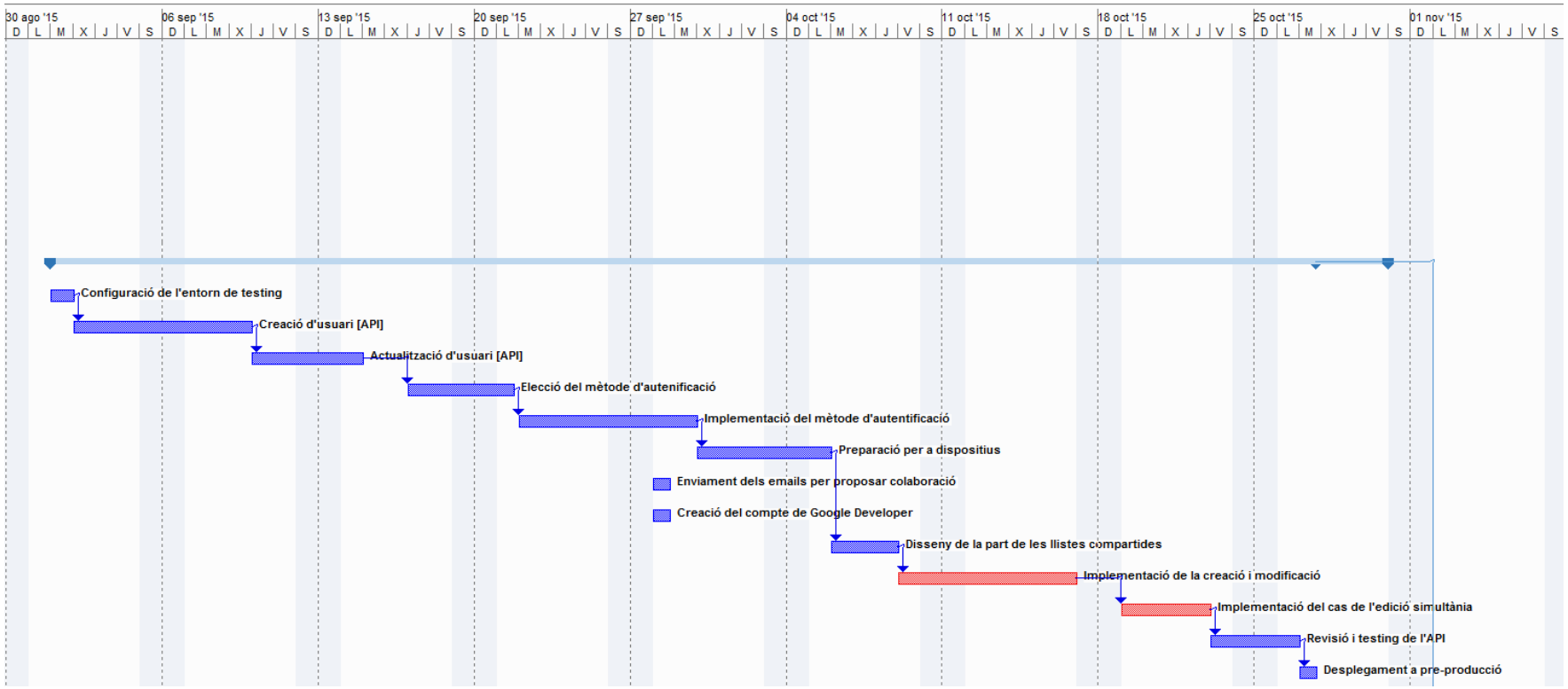

Figure 17: Gantt chart of the stage API development

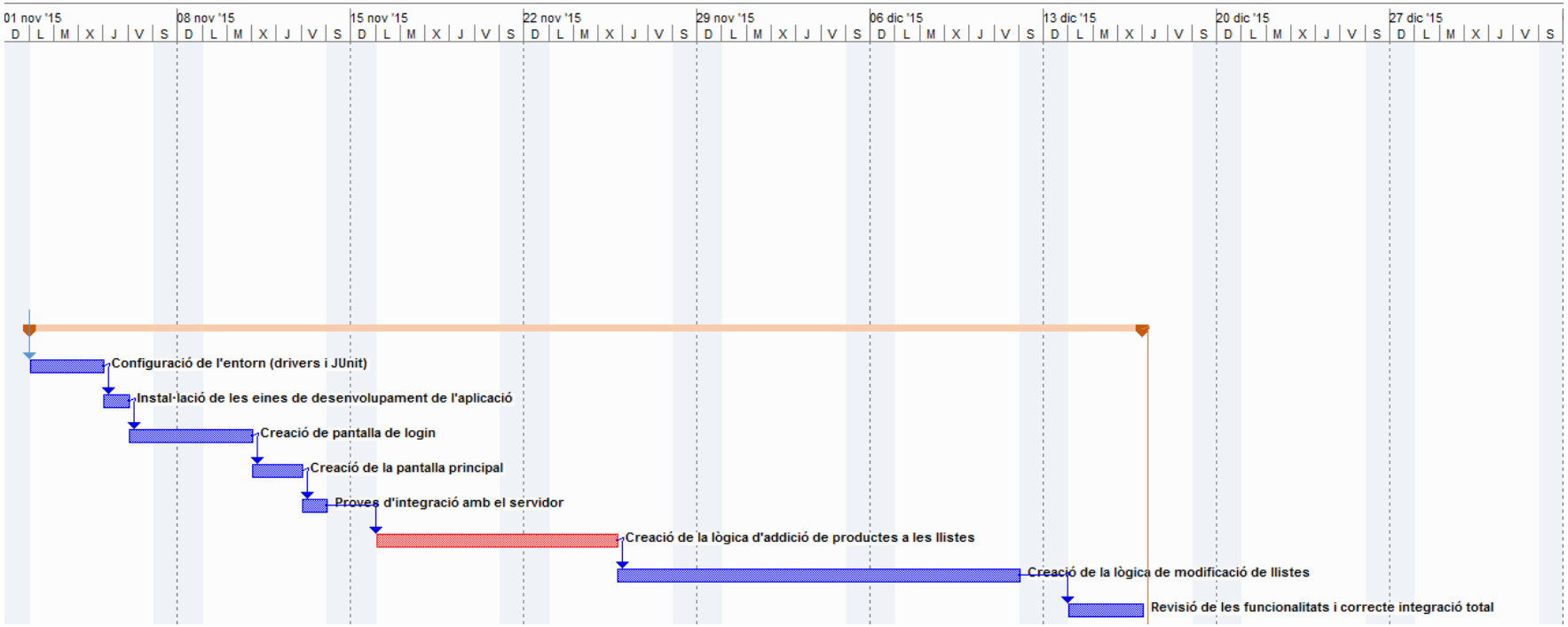

Figure 18: Gantt chart of the stage App development

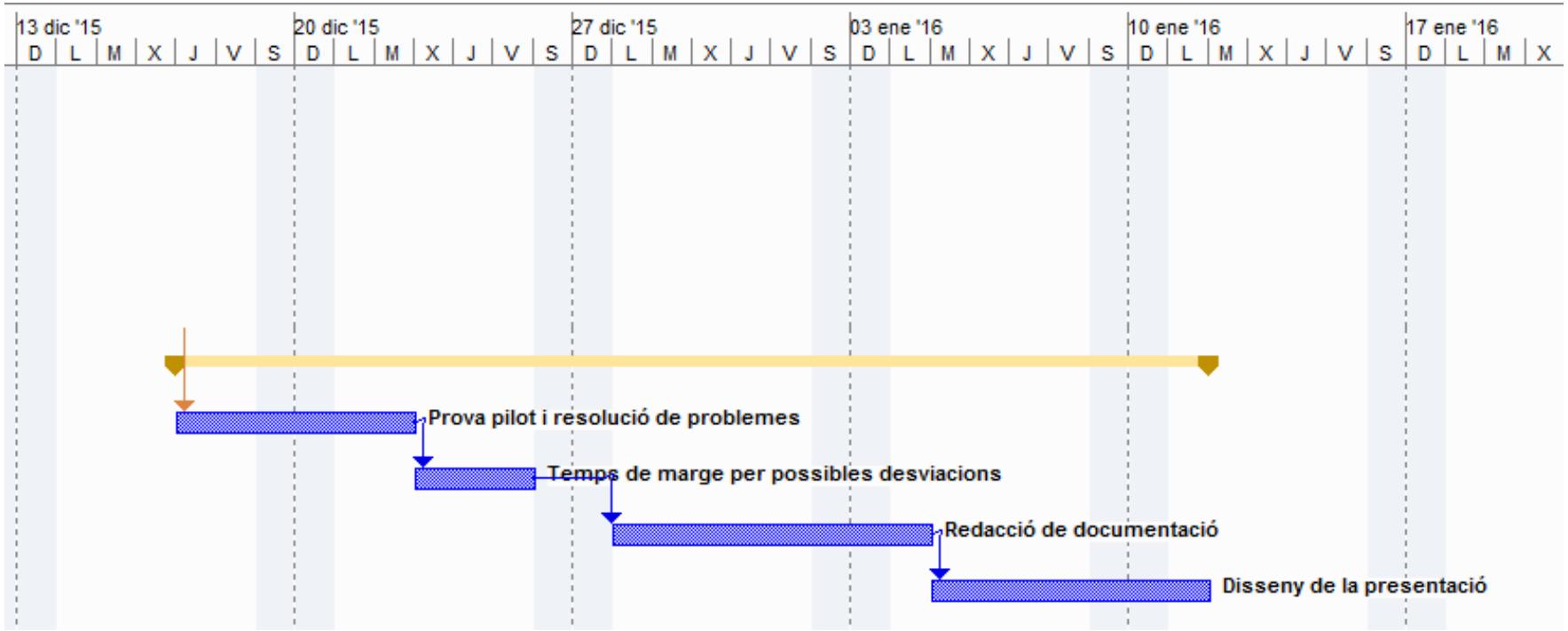

Figure 19: Gantt chart of the stage Final delivery and documentation

D Gantt chart of project development

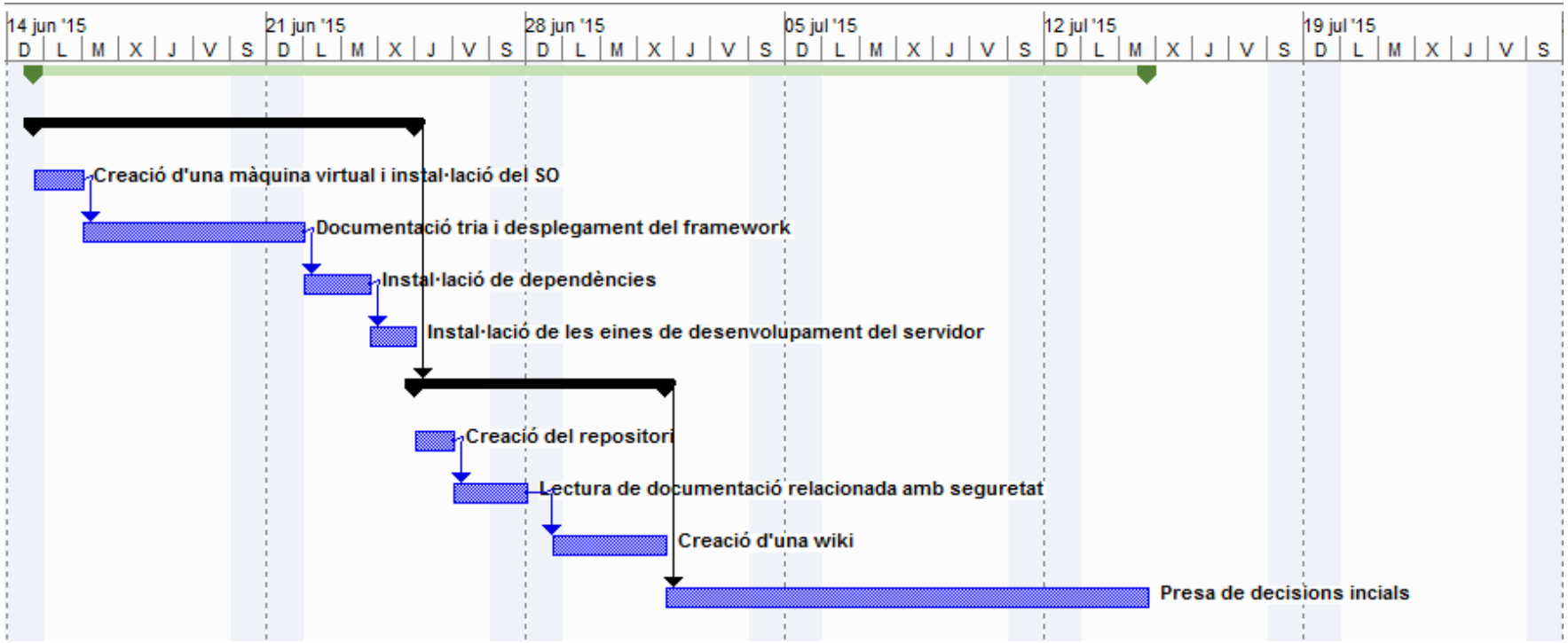

Figure 20: Gantt chart of the stage Preparation and evaluation of the project

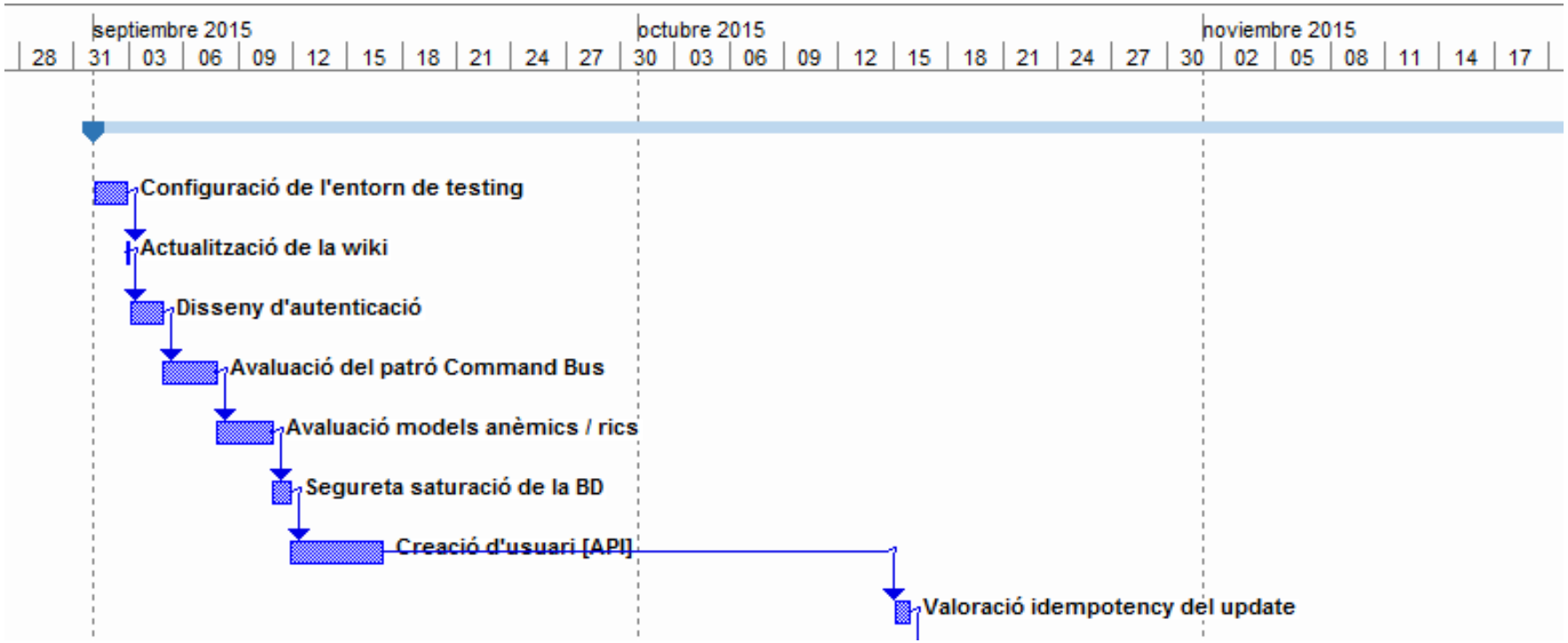

Figure 21: Gantt chart September - October

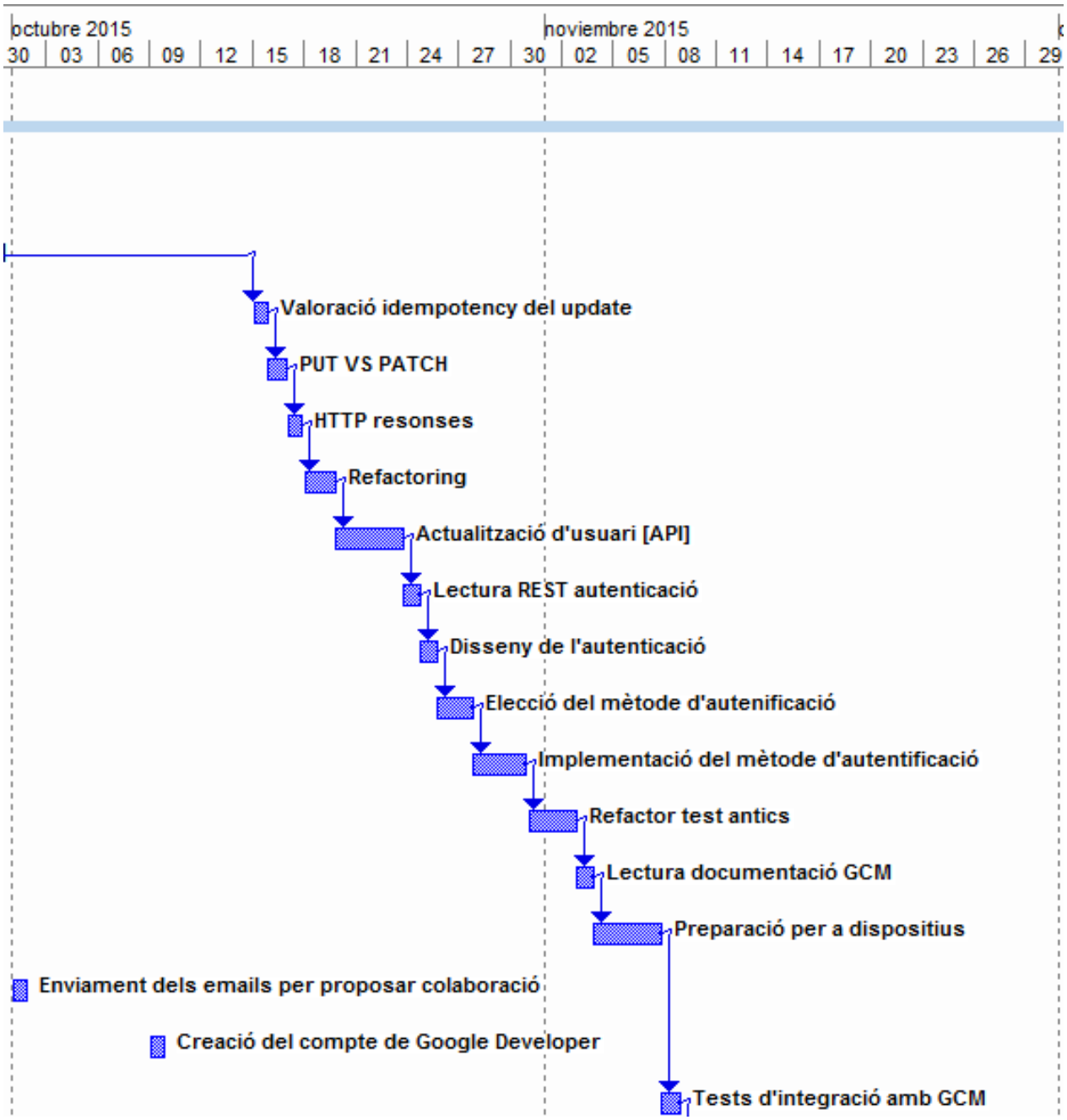

Figure 22: Gantt chart October - November

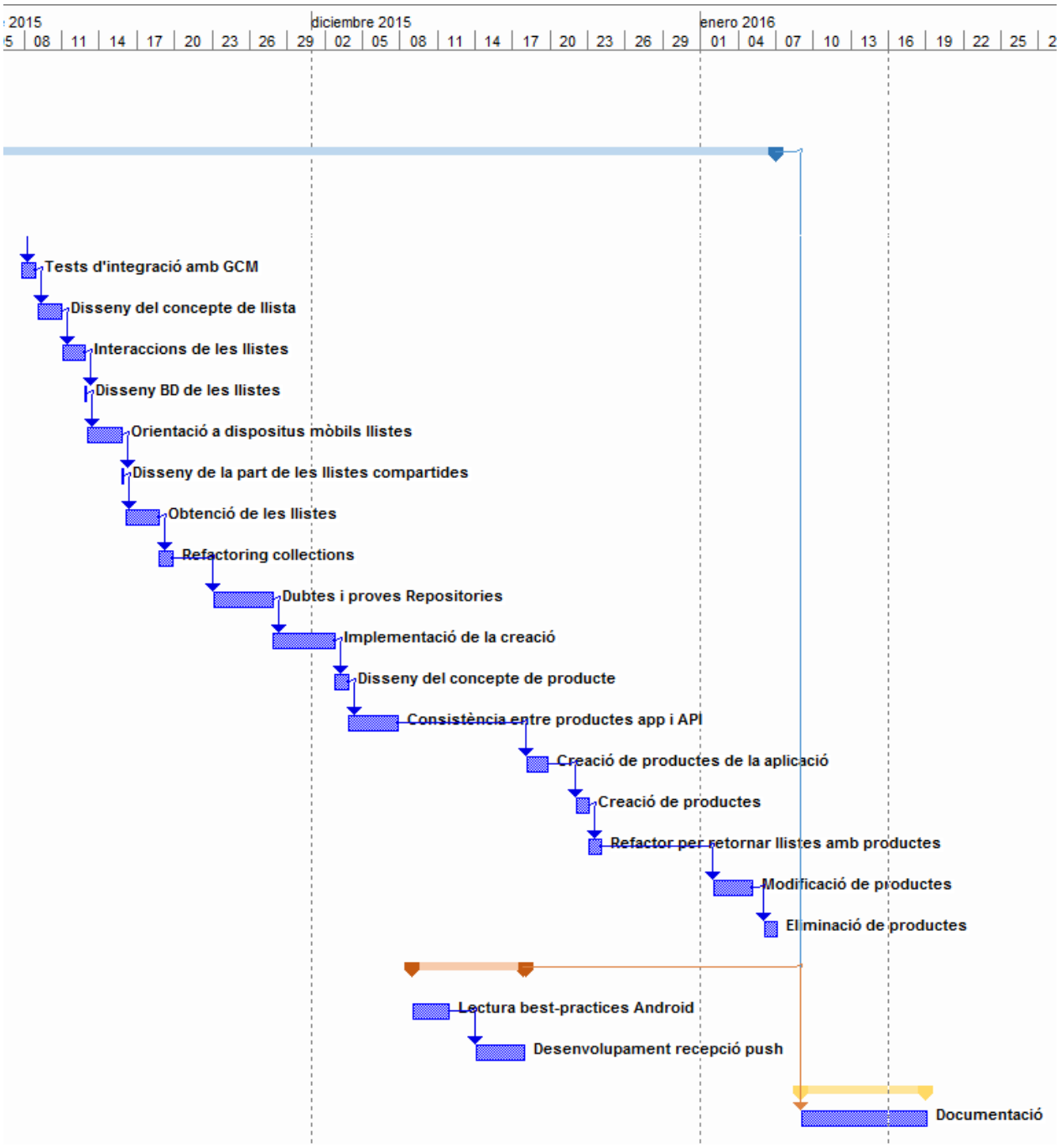

Figure 23: Gantt chart November - January

E Resources and tasks according to the initial planning

Nom	Duració	Nom dels recursos	Cost
Preparació i avaluació	**136 hrs**		**3.505,00 €**
Creació d'una màquina virtual i instal·lació del SO	8 hrs	Desenvolupador júnior	160,00 €
Documentació tria i desplegament del framework	30 hrs	Arquitecte júnior[50%]	450,00 €
Instal·lació de dependències	8 hrs	Desenvolupador júnior	160,00 €
Instal·lació de les eines de desenvolupament del servidor	8 hrs	Desenvolupador júnior	160,00 €
Creació del repositori	4 hrs	Desenvolupador júnior	80,00 €
Lectura de documentació relacionada amb seguretat	12 hrs	Arquitecte júnior	360,00 €
Creació d'una wiki	16 hrs	Desenvolupador júnior	320,00 €
Presa de decisions inicials	66 hrs	Arquitecte júnior[50%] / Analista júnior[50%]	1.815,00 €
Desenvolupament API	**156 hrs**		**3.680,00 €**
Configuració de l'entorn de testing	4 hrs	Desenvolupador júnior	80,00 €
Creació d'usuari [API]	24 hrs	Desenvolupador júnior	480,00 €
Actualització d'usuari [API]	12 hrs	Desenvolupador júnior	240,00 €
Elecció del mètode D'autenticació	10 hrs	Arquitecte júnior[50%] / Analista júnior[50%]	550,00 €
Implementació del mètode D'autenticació	24 hrs	Desenvolupador júnior	480,00 €
Preparació per a dispositius	16 hrs	Desenvolupador júnior[50%] / Arquitecte júnior[50%]	400,00 €
Enviament dels emails per proposar col·laboració	2 hrs	Analista júnior	50,00 €
Creació del compte de *Google Developer*	2 hrs	Desenvolupador júnior	40,00 €
Disseny de la part de les llistes compartides	12 hrs	Arquitecte júnior	360,00 €
Implementació de la creació i modificació	24 hrs	Desenvolupador júnior	480,00 €
Implementació del cas de l'edició simultània	16 hrs	Desenvolupador júnior	320,00 €
Revisió i testing de l'API	8 hrs	Desenvolupador júnior	160,00 €
Desplegament a pre-producció	2 hrs	Desenvolupador júnior	40,00 €

Nom	Duració	Nom dels recursos	Cost
Desenvolupament Aplicació	**132 hrs**		**2.790,00 €**
Configuració de l'entorn (*drivers* i *JUnit*)	12 hrs	Desenvolupador júnior	240,00 €
Instal·lació de les eines de desenvolupament de l'aplicació	4 hrs	Desenvolupador júnior	80,00 €
Creació de pantalla de login	12 hrs	Desenvolupador júnior	240,00 €
Creació de la pantalla principal	8 hrs	Desenvolupador júnior	160,00 €
Proves d'integració amb el servidor	4 hrs	Desenvolupador júnior	80,00 €
Creació de la lògica d'addició de productes a les llistes	30 hrs	Arquitecte júnior[25%] / Desenvolupador júnior[75%]	625,00 €
Creació de la lògica de modificació de llistes	50 hrs	Arquitecte júnior[25%] / Desenvolupador júnior[75%]	1.125,00 €
Revisió de les funcionalitats i correcte integració total	12 hrs	Desenvolupador júnior	240,00 €
Entrega Final i documentació	**72 hrs**		**1.480,00 €**
Prova pilot i resolució de problemes	16 hrs	Desenvolupador júnior[50%] / Analista júnior[50%]	360,00 €
Temps de marge per possibles desviacions	12 hrs	Desenvolupador júnior	240,00 €
Redacció de documentació	24 hrs	Desenvolupador júnior	480,00 €
Disseny de la presentació	20 hrs	Desenvolupador júnior	400,00 €

Table 13: Analysis of the human costs of the project